TELECOM
PARIS

Thèse

présentée pour obtenir le grade de docteur de l'École Nationale Supérieure des Télécommunications

Spécialité : Informatique et Réseaux

Arnaud Bailly

Du requis au garanti : contrats dans un calcul d'objets temporisés mobiles.

Soutenue le 17 Décembre 2002 devant le jury composé de

Gérard Florin	Président
Jean-Pierre Courtiat	Rapporteurs
Alessandro Fantechi	
Isabelle Demeure	Examinateurs
Kathleen Milsted	
Jean-Bernard Stefani	
Elie Najm	Directeur de Thèse

Remerciements

Je tiens tout d'abord à remercier Elie Najm, qui a été mon directeur de thèse, et qui m'as toujours prodigué son soutien de manière inconditionnelle dans les moments les plus difficiles, et offert son amitié (et un verre de bière en bien des occasions). Pour tout cela je resterai éternellement redevable et reconnaissant envers lui.

Je remercie également Jean-Bernard Stefani et Kathleen Milsted pour avoir accepté de participer au jury de ma soutenance, et pour avoir, en tant que responsables succesifs de l'encadrement de mes travaux pour la section DTL/ASR à France Télécom R&D, permis le financement de la majeure partie de cette thèse.

Je remercie Alessandro Fantechi pour avoir accepté de rapporter cette thèse, et pour les nombreuses discussions que nous avons pu avoir lors de ses visites en France et de nos rencontres de part le monde.

Je remercie aussi chaleureusement Jean-Pierre Courtiat pour avoir rapporté mes travaux, pour sont humour et sa précision dans les commentaires qu'il a pu me prodiguer.

Je remercie bien sûr les professeurs, doctorants, et personnels du département informatique et réseaux de l'ENST, qui ont participé à créer une ambience de travail agréable et stimulante durant ces longues années. Parmis les nombreux doctorants que j'ai pu croiser, et sans volonté d'hexaustivité ni ordre particulier, je tiens à citer les docteurs Wessam Ajib, Didier Verna, Nadine Richard, Abbas Ibrahim, Arnaud Fontaine, Philippe Martins, Muhammad Kazmi, Dany Zebiane, Rani Makke, Pascal Moyal, Alexandre Tauveron, Cyril Carrez, Jean-Pilippe Démoulin, et Robert Bestak. À eux s'ajoutent les plus amicaux participants des «Zappy» du Vendredi et d'autres événements, nommément Abdelkrim Nimour, Arnaud Février, Gaël Chardon, Odile Derouet, Jean Leneutre, Sonia Heimann et Vania Conan, qui ont souvent supporté mes désidérata neuronaux et mes jugements à l'emporte-pièce. Je les en remercie tous bien chaleureusement.

Je remercie tout particulièrement certaines personnes que j'ai eu la chance de croiser, avant ou pendant ma thèse, et qui me maintiennent encore, et ce de manière complètement déraisonnée et incompréhensible, leur soutient démesuré et leur amitié bienveillante. Comme j'ai beaucoup de chance elles sont nombreuses, et leur mérite n'a pas d'égal; aussi elles ne sauraient être comparés entre elles. Je remercie donc Tatiana Aubonnet, Loutfi Nuaymi, Cyril Lacaud, Sophie Diallo, Frédéric Fourneau, Richard Sanders et Jorunn Dugstad. Que nos chemins ne se séparent pas.

Enfin, je remercie mes parents, sans qui tout cela n'aurait pu arriver...

Résumé

Introduction

Les systèmes répartis exhibant des contraintes temporelles sont aujourd'hui présents partout: de bons exemples sont les systèmes de télécommunications numérique ou bien les systèmes embarqués comme rencontrés dans l'électronique grand public ou l'avionique. La conception et le déploiement de tels systèmes sont intrinsèquement complexes. De nombreuses méthodes de développement fondées sur l'analyse formelle de ces systèmes ont donc été proposées. Ces méthodes sont basées sur la description des comportements possibles du système analysé.

Nous soutenons toutefois que les méthodes existantes sont souvent inadaptées car elle ne permettent pas de distinguer les actions que le système *peut* produire des actions qu'il *doit* produire. L'absence de ce pouvoir de distinction conduit à des spécifications purement factuelles, où raisonner à propos d'un système ne peut être accompli que si l'on connaît absolument son environnement, ce dernier pouvant produire des actions interférant avec celles du système étudié.

Nous proposons donc des méthodes de description et d'analyse fondées sur la reconnaissance de cette distinction. La description d'un système indique alors les exigences (ou suppositions, ou *assumptions*) faite pas le système sur son environnement, ainsi que les *garanties* respectées par le système quand son environnement se comporte de manière adaptée. L'ensemble forme un *contrat* entre le système et son environnement.

Nous utilisons pour langage une extension temporisée du π-calcul pour objets mobiles. Nous proposons une notion d'équivalence pour les termes de ce langage fondée sur la bisimulation, ainsi qu'un système de preuve permettant de décider de manière exacte et complète si deux termes à états finis se comportent de manière identique. Cependant, les descriptions utilisant le π-calcul produisent en général un nombre infini d'états.

Nous proposons donc une autre méthode d'analyse, fondée sur le typage comportemental, qui permet de prendre en compte de telles descriptions. Notre système de type nous permet de raisonner de manière *compositionelle*: chaque élément forme alors un *composant*, et les composants ne peuvent être assemblés que si chacun d'entre eux ne fait pas de suppositions excessives sur le comportement des autres composants. Vérifier qu'un module se comporte de manière correcte envers son environnement se réduit à vérifier qu'il est bien typé. L'analyse de types étant statique (à la compilation et non à l'exécution) cela permet de savoir si un module est correct dès sa conception.

Enfin nous remarquons que les applications temps-réel dépendent très fortement de leur environnement d'exécution, et la vérification statique ne sert à rien si la machine d'exécution d'un programme (système d'exploitation, machine virtuelle) peut se retrouver dans l'incapacité d'allouer les ressources demandée dynamiquement. Pour cela, il nous parait intéressant d'utiliser les résultats obtenus par typage pour garantir l'exécution correcte des applications. Une machine d'exécution peut en effet utiliser les informations contenues dans les types comportementaux pour prévoir les besoins en ressources système (deadlines, bande passante) et pratiquer un contrôle d'admission à l'entrée. Nous remarquons qu'une preuve de bon typage étant une preuve de bon comportement, elle peut également servir à contrôler le code lui-même, en tant que «proof-carrying code», comme pratiqué actuellement par la machine virtuelle Java. Une application évidente de nos travaux est alors la prévention des «Denial of Service attacks». Nous montrons pour conclure que les notations et méthodes définies sont utilisables dans le cadre de développements utilisant la notation UML-RT.

Le langage π^δ

Nous définissons un calcul de processus temporisés permettant de représenter les contraintes exhibées par des composants temps-réel dont le code peut migrer dynamiquement sur un réseaux d'ordinateurs. Chaque processus comprend un terme algébrique t et un contexte ρ donnant une valeur à des horloges dont la valeur évolue au cours du temps; l'ensemble forme un terme de π^δ, noté tout simplement $t\rho$.Les termes t respectent la syntaxe suivante, en prenant a, b, c appartenant à un ensemble infini de noms de *ports*, et x, y, z appartenant à un ensemble infini de noms d'*horloges*:

$$t ::= 0 \mid Error^{\upsilon} \mid [\sigma,\, \upsilon]\pi.\, t \mid [a = b]t \mid \nu n\ t \mid t + t \mid t|t \mid A(\tilde{n})\ .$$

où n peut être indifféremment un nom de port ou d'horloge. Les processus de terme 0 sont appelés *processus passifs*; les processus de terme $Error$ sont appelés *processus bloquants* (nous dirons aussi informellement qu'ils sont indésirables, ou qu'ils produisent une erreur). Les opérateurs qui apparaissent dans la syntaxe ci-dessus sont usuellement nommés, de gauche a droite: préfixe, comparaison, restriction, somme (ou *choix*), composition (ou *parallèle*), et instanciation.

La syntaxe des actions réductibles est donnée ci-dessous:

$$\pi^\star ::= a(n) \mid \overline{a}\, n \quad \text{et} \quad \pi ::= \tau \mid \pi^\star\ .$$

L'action silencieuse τ peut-être vue comme une action *interne* au processus qui l'effectue. Tous les autres préfixes ont un *sujet* a appartenant à $\mathbf{P}$, et un *objet* n appartenant à $\mathbf{N}$. Informellement, $\overline{a}\, n$ décrit une *émission* sur le port sujet a d'un message contenant l'objet n, et $a(n)$ décrit la réception d'un message sur le port a, dont l'objet reçu est appelé n dans la continuation t du processus.

Les contraintes de *sélection* σ et les conditions d'*urgence* υ (ou *deadlines*) respectent la syntaxe suivante, donnée respectivement par

$$\sigma ::= \sigma \wedge \sigma \mid x \sharp^{\sigma} q \mid x - y \sharp q \quad \text{et} \quad \upsilon ::= \upsilon \wedge \upsilon \mid x \sharp^{\upsilon} q \mid x - y \sharp q$$

où $\sharp \in \{<, \leq, \geq, >\}$, $\sharp^{\sigma} \in \{<, \leq, \geq\}$, $\sharp^{\upsilon} \in \{<, \geq\}$, et q est une valeur décrivant un instant précis dans la marche temporelle. Le processus $Error^{\top}$, où $\upsilon = \top$ est la condition toujours vraie, est $Error$ par simplicité.

Le point essentiel est que la sémantique de ces processus décrit ce que chacun d'entre eux *offre* et *exige*. Ainsi, un processus présentant un préfixe dont la contrainte σ est vérifiée sans que υ soit vérifiée fait une offre à sont environnement. L'environnement est alors libre d'accepter l'offre et de se synchroniser avec le processus offrant, ou bien de refuser. Dans ce dernier cas, le temps peut s'écouler librement. sans qu'aucune action ne se passe. Si toutefois l'action est urgente (dont υ) est vraie, alors l'environnement *ne peut refuser* la synchronisation. S'il ne propose a temps une offre correspondant à l'exigence présentée par le processus, alors ce dernier entraîne tout le système vers une erreur. Les règles de sémantique opérationnelle pour π^{δ} sont données en Tables 3.2 et 3.3.

L'axiomatisation et le système de types

Présenter ici les détails techniques liés à l'axiomatisation ou aux système de types serait déplacé, car leur complexité est trop importante. En place de cela, nous en décrivons les principes. Ces deux contributions techniques reposent sur une *sémantique symbolique* du langage π^{δ}.

La sémantique symbolique

Un problème essentiel posé par l´étude des systèmes temps réels est la nature *continue* du temps. En effet, un temps continue implique que laisser passer le temps peut aboutir à *une infinité* de processus différant par la valeur de leurs horloges. Rajeev Alur et David Dill ont montré cependant que, lorsque les valeurs temporelles apparaissant dans les contraintes des termes sont *rationnelles*, alors qu'une infinité de transitions laissant passer le temps pour un processus peuvent en fait être représentées par *une seule* transition [AD94]. Une telle transition est dite *symbolique*, ou *abstraite*. Les processus reliés par des transitions symboliques sont appelés *régions*, ou *zones* (une zone contient plusieurs régions). Un parle aussi de *graphe des régions*.

L'abstraction proposée par les régions rend de nombreux problèmes décidables pour les processus temporisés dont l'abstraction est à états finis (*i.e.* la seule source «d'infinitude» provient du temps, le contrôle de ces automates restant à états finis). Cependant, nos processus incluant des termes au moins aussi expressifs que le π-calcul, leur partie de contrôle peut atteindre un nombre infini d'états (le *pi*-calcul permet d'encoder le λ-calcul [SW01]). C'est la génération de noms (*i.e.* l'introduction de restrictions) en nombre toujours croissant qui permet à des termes d'atteindre l'infinité en espace. Nous utilisons donc une deuxième forme d'abstraction, celle ci définie par Hennessy et Lin [HL95] permettant d'abstraire la géneration de noms et de la réduire à un fragment finitaire. Cette abstraction permet toujours au processus de générer un espace d'états infini, mais elle permet d'effectuer la comparaison entre processus non sur la base des noms qu'ils génèrent (ce qui est infaisable pratiquement), mais sur la base des comparaisons quíls font entre ces noms (ce qui est fait dans le système de preuves).

Le système de preuves

Nous avons donc adapté des systèmes de preuve existant [LY02, Lin98], pour permettre de prouver de manière cohérente, valide et complète, que deux processus sont liés par une relation d'équivalence appelée *bisimilarité temporisée tardive*, qui est une simple extension de la notion de bisimilarité introduite par Milner pour CCS et le π-calcul.

Notre adaptation est cependant non trivialle car une propriété temporelle usuellement respectée ne l'est pas par nos processus, de part leur nature contractuelle représentant des offres et des exigences. En quelques mots, nous devons découper l'espace d'états de nos processus de manière suffisemment fine, ce qui requiert de surcroit d'effectuer des correspondances assez élaborées entre partitions. La complétude du système de preuves est obtenue en permettant d'interrompre la comparaison de différentes branches d'un choix lorsque l'une des branches génère une erreur (et donc interdit d'effectuer toute transition). Ainsi, des processus qui se comporte de manière équivalente avant le bloquage mais auraient pu se comporter de manière différente après sont quand-même identifiés (comme il se doit). Le système de preuves, ainsi que la sémantique symbolique des processus, est décrite au Chapitre 4.

Le système de types

Alors que les preuves d'équivalence ne sont faisables que sur des processus à états finis (nous n'autorisons pas l'usgae de la récursion), le système de types permet d'analyser des configurations atteignant un nombre d´etats infini. Ceci peut être accompli en effectuant une abstraction supplémentaire sur les noms de ports qui sont créés ou transmis. La sémantique symbolique étant sur ce point plus précise que nos types, nous pouvons l'utiliser pour mener à bien l'analyse des types, résolvant de ce fait les problèmes liés à la nature continue du temps de la même manière que précédemment. Une contrainte forte (mais réaliste) est cependant introduite sur la duplication des noms de ports dans une configuration bien typée: si la forme (*i.e.* le type) des communications sur un port donné peut évoluer au cours du temps, alors il ne devra pas y avoir plus de deux exemplaires de ce nom dans lénsemble du systèmes: les interactions se font par paires, uniquement par des processus qui sont de cette manière «en vis-à-vis».

Les résultats obtenus sont l'absence de bloquage dû à un manque de synchronisation entre processus (le contrat exhibé par un processus n'est jamais rompu), et l'absence de bloquage de la progression du temps (*i.e.* le système continue à évoluer, évitant le paradoxe d'Achille et de la tortue décrit par Zénon d´Élée), sous une condition d'équité faible concernant le transitions silencieuses (appelées aussi transitions internes).

Les Applications

Nous commençons le chapitre des applications par un peu de litérature comparée autour de la notion de contrat (en logique, essentiellement) et de son utilisation en informatique. Nous établissons un lien en particulier entre la logique déontique telle

que décrite par Von Wright [von51] et nos systèmes à transitions étiquetées portant les offres et les exigences d'un processus.

Ensuite, nous avançons une utilisation possible de π^δ comme *langage de modèle* pour des plateformes d'exécution temps-réel distribuées. Certaines propositions récentes [MNCK99] vont en effet dans le sens de distinguer des objets "parfaits" ou objets de modèle, qui représentent l'exécution ideale, telle qu'elle devrait se produire, des objects *actifs*, qui effectuent réellement les tâches calculatoires à accomplir. Les ojets de modèle donnent la structure, la référence de l'application à exécuter. Alors, des ojets de modèles exprimés en $\mathcal{A}$rtOC fourniraient un support sémantique précis à la plateforme et au programmeur devant concevoir l'application. À l'exécution, une incompatibilité entre le modèle de l'application et les activités observées de cette application pourraient amener à prendre des mesures contre celle-ci. Cette proposition se rapproche du model-carrying-code [SRRS01].

Enfin, nous présentons un langage d'acteurs appelé $\mathcal{A}$rtOC, que nous avons également développé, et dont les principes sont similaires à π^δ. Ce langage est toutefois plus proche des langages de programmation classique, et les communications entre agents se font par passage de message asynchrone, alors que les communications sont synchrones dans π^δ. Cette dernière hypothèse est plus conforme à la réalité, ce qui rapproche le langage de la pratique usuelle, et devrait réduire le temps d'apprentissage moyen pour en arriver à la maîtrise par rapport à π^δ. Nous montrons finalement comment des concepts usités (syntaxiquement) dans des programmes $\mathcal{A}$rtOC trouvent un pendant naturel dane le langage de specification UML-RT. Ceci nous mêne a proposer une intégration de programmes $\mathcal{A}$rtOC dans un processus de développement de systèmes temps-réel utilisant la notation UML-RT.

Conclusion

Nous avons étudié la notion de *contrat* entre objets répartis et temps réel. Cette étude nous apermis de définir formellement un cadre sémantique permettant notemment de comparer notre approche avec de nombreux travaux existants, où la notion de contrat, si elle n'est pas toujours explicite, se trouve mise au jour par notre comparaison. Nous avons également tenté de donner une portée pratique à nos travaux. Si les aspects techniques abordés ont été très satisfaisants de part leur complexité, il est toutefois clair que cette complexité se retrouve au niveau des modèles manipulés. Un défi pour l'avenir serait donc de simplifier certains côtés de l'approche pour en tirer le meilleur, et fournir au programmeur potentiel des outils plus facilement manipulables que ceux qui pourraient être obtenus avec la théorie actuelle. C'est certainement sur ces points pratiques que les contributions de nos travaux sont les plus faibles, et y remédier pourrait s'avérer un chemin d'étude de longueur au moins égale à celle du chemin déjà parcouru.

Contents

1 Introduction **15**

2 Open Distributed Processing and Real-Time **19**
2.1 Open Distributed Processing 19
2.1.1 Overview 19
2.1.2 The Computational Viewpoint 20
2.1.3 The Engineering Viewpoint 22
2.1.4 Specifications, Viewpoints and their Relations 23
2.2 Modeling and Verifying Real-time Systems 23

3 An Untyped π^δ: Syntax and Semantics **25**
3.1 Introduction 25
3.2 The Syntax: Processes, Ports, and Clocks 27
3.3 The Time Domain and Its Properties 29
3.4 Clock Valuations, Constraints, And Satisfaction 30
3.5 Well-Formedness Criteria for Processes 31
3.6 The Semantics: Labeled Transition Systems for Processes 32
3.7 Semantics of A Small Example 37
3.8 Model Properties Over Time 40
3.9 Discussion and Related Works 41

4 A Proof System for π^δ **45**
4.1 Process Equivalence and Abstract Semantics 45
4.1.1 Axiomatizing Dense Time 47
4.1.2 Axiomatizing Name Instantiation 48
4.1.3 Manipulating Zones, Regions, and Matching Constraints . . . 49
4.1.4 Abstract Labeled Transition Systems 54
4.1.5 Semantics of a Small Example (Continued) 61
4.2 Algebraic Laws for Processes 63
4.2.1 Relating Concrete and Abstract Transition Systems 63
4.2.2 Symbolic Timed Late Bisimulation 64
4.2.3 A Proof System for Terms with Finite Control 66
4.2.4 Soundness and Completeness 73
4.3 Conclusion 73

5 A Behavioral Type System for π^δ 77
5.1 Introduction . . . 77
5.2 The Type Language . . . 78
5.2.1 The Syntax for Types . . . 79
5.2.2 Type Semantics . . . 80
5.2.3 Discourse on The Meaning of Types . . . 83
5.3 Type Equivalence, Subtyping, and Polymorphism . . . 85
5.4 Type Checking . . . 89
5.4.1 Restraining the Expressive Power of Processes . . . 89
5.4.2 Type Compatibility and Context Composition . . . 91
5.4.3 The Type System . . . 94
5.5 Properties of our Type System . . . 97
5.6 Liveness and Composition . . . 99
5.7 Related Works and Conclusion . . . 101
5.7.1 Behavioral Types . . . 101
5.7.2 Assume/Guarantee Reasoning . . . 103
5.7.3 Conclusion . . . 103

6 Applications 105
6.1 Introduction . . . 105
6.2 Contracts in Logic and Computation . . . 105
6.2.1 Contracts in Deontic Logic . . . 105
6.2.2 Contracts in Behavioral Types . . . 106
6.2.3 Contracts in Hoare Logic and Its Extensions . . . 107
6.2.4 Contracts in π^δ . . . 108
6.2.5 Contracts Elsewhere in Computing . . . 109
6.3 Provable QoS Support for Middleware Platforms . . . 110
6.3.1 QoS Specification . . . 111
6.3.2 Model Components and Type Checking . . . 112
6.4 Integration with UML-RT . . . 113
6.4.1 $\mathcal{A}$**rtOC**: An actor-like, asynchronous π^δ . . . 113
6.4.2 A Meta-Architecture For UML-RT . . . 116
6.4.3 Notions of $\mathcal{A}$**rtOC** in UML . . . 117
6.4.4 Relationships with UML-RT and Examples . . . 121
6.5 Conclusion . . . 125

7 Conclusion 127

A Proofs 129

List of Figures

2.1 Communicating roles of an Operational Service u 21
2.2 An example of communicating interfaces, step (1) 21
2.3 An example of communicating interfaces, step (2) 21
2.4 An ODP Configuration at Engineering Level 23

4.1 Operations on 2-Polyhedra 51

6.1 The Syntax of *ArtOC* 114
6.2 The Relationships Among UML-RT Entities 116
6.3 An UML-RT Collaboration Diagram 117
6.4 The "Stack of Models" 118
6.5 The Relationships among *ArtOC* Meta-Architectural Concepts 119
6.6 The Interface Type Hierarchy 120
6.7 The Contract Type Hierarchy 121
6.8 Stereotypes for *ArtOC* Specifications 122
6.9 Realization of UML-RT specifications by *ArtOC* Programs 123
6.10 A Video Transmission Example in *ArtOC* UML notation 124

Chapter 1

Introduction

If one wishes to classify today's omnipresent computational devices he or she may not afford to miss, without ruining the relevance of his or her study, the class of artefacts that exhibit a time-constrained behavior. Examples rise in very dissimilar places and environments, to name a few: embedded systems such as electronic houseware (multiparted high-fidelity systems, television set-top boxes) or avionic sytems (telemetric devices, guiding devices), telecommunication systems (network switches, voice and video transportation using data-oriented communication protocols), control systems for nuclear plants and railroads, and finally computer games and graphic animation.

We are interested in real-time software engineering. Clearly, from the variety of systems that are mentioned above, one may guess that a tremendous number of very different proposals have been made in this domain to master program specification, design, development, testing and execution. To trace a rough outline however, software assurance still relies mainly on empirical, ad-hoc techniques for the first four steps, while execution correctness is ususally maintained by over-provisioning of resources. In a general fashion, the more an application is unpredictable, the less solutions exist to easily create it, and the more over-provisioning should be done. Yet, *flexible* real-time applications, which are inherently unpredictable, are now required to be created as witnessed by the recently issued Object Management Group (OMG) and Unified Modeling Language (UML) proposals.

To cope with these problems, we advocate the use of formal methods in each and every life-cycle step of flexible real-time applications; this should improve the confidence one may have in them. However, not being ambitious enough to propose a general method, we restrict our target: we consider only applications with hard real-time requirements, and we propose analysis methods for non-probabilitic models that only issue $true/false$ answers (no probabilities, like *e.g.* "40% correct"). Furthermore, we take the "flexible applications" to be "open distributed applications" in the sense of the Open Distributed Processing Reference Model (RM-ODP) [ODP95], which have many implications that we now detail.

Distribution is indeed a major concern that we may not avoid, as illustrated by many of the industrial applications mentioned above. A distributed system is said to be open [ODP95] if it may be extended, updated or reconfigured at run-time. To have *extendable* systems stresses the need of creating inter-operable software mod-

ules that are able to perform in inaccurately known environments while preserving their individual properties, hence allowing easy *composition*. In turn, to be able to *update* some components requires a notion of *refinement* (it is sometimes also called *implementation*, and its dual notion is *abstraction*) defining the conditions allowing some refined component to replace another one while preserving the supposedly correct behavior of the whole system. Finally, *reconfigurability* implies that the communication topology among given components may be modified, for example existing links may fall, or new ones may be established.

Object-Oriented Concurrent Programming (OOCP) provides the structuring concepts that we need. An *object* is a computational entity that encapsulates its own data and behavior. Each object may interact with the outside world through one or many *interfaces*. When interfaces are sufficiently well-defined, an object may be viewed as an independant software module and compositional techniques may be applied easily. Furthermore, *interface subtyping* and *polymorphism* allow refinement to be performed, while *object references* can be passed around among objects, simulating reconfiguration.

Numerous formal theories for compositionality and refinement have been devised during the last twenty years, many of them coping with concurrent objects either directly or through faithful encodings. A large body of work independently concerns real-time aspects of software. Surprisingly, though informal or applicative papers dealing with both aspects have been published, we know none but very few such proposals using formal methods.

We therefore propose to alleviate this lack by providing a timed process algebra derived from the π-calculus [MPW92]. We chose the π-calculus because it has a thoroughly developped and well-studied theory of expressiveness, equivalence, object encoding and implementability issues. We show how to deal with some of these issues in a timed setting: we give an appropriate notion of equivalence between objects, axiomatic proof rules to decide equivalence over finite terms, and a static analysis (type-theoretic) method to check that objects of some potentially infinite-state configuration are well-behaved regarding one another. This method naturally leads to view objects as bound by *contracts*, their interfaces stating which *role* they should play in each contract.

Direct applications are proposed for concurrent object-oriented modeling and QoS-enforcing middleware engineering. We show that our concepts are actually verysimilar to the ones introduced by the UML-RT [GBSS98, Sol97, RS] proposal. This is interesting since [RS] explicitly calls for object (*i.e.* contract) compatibility decision procedures. Our application to middleware is based on informal existent proposals [Sch00, MNCK99] for QoS-enforcing, real-time execution support. We propose to use our language for application *coordination* purposes [GC92], as done by the "model components" of [MNCK99]. Then, a program asking to be executed may either carry a proof of its correctness obtained by anterior verification (the proof can be checked by the platform [NL96]) or be type-checked on location as in the Java Virtual Machine (JVM) [Mic95].

We now give a more detailed, chapter by chapter, account of this thesis.

The first chapter after this introduction recalls the essential concepts defined in RM-ODP. This constitutes a convenient repository of concepts over which we shall

elaborate many of our later developements.

In Chapter 3 we introduce our calculus for timed processes with synchronous interactions, that we call π^δ. We propose an operational semantics for it that yield labelled transition systems with which timed computations can be represented. The operators of our language are strictly more powerful than the operators present in all timed process algerba that we know of. This brings us to propose an original model for processes, allowing them to distinguish *mandatory* from *possible* actions. In case a component should fail to provide the required behavior, an explicit contract violation occurs, stopping the system. We obtain a calculus in which we are able to directly express *contracts* between components.

This should be seen as a natural introduction of deontic logic [MW93] concepts into process algebras. Previous works in the converse direction have been accomplished by Fiadeiro and Maibaum in an untimed temporal logic setting [FM91], and by Dignum and Kuiper in a timed temporal logic setting [DK97].

In Chapter 4 we propose a proof system deciding a form of late timed bisimulation equivalence for finite terms of our calculus. In order to do that, we must devise a *symbolic* (also called *abstract*) semantics for the terms of our calculus. This semantics is largely the result of a combination of two symbolic bisimulations [HL95] proposed for the π-calculus [Lin94] and for a calculus of timed automata [LY00]. In a nutshell, it combines two abstractions: an abtraction of clock values (which are uncountably-many), and an abstraction on port names of the π-calculus (which are infinitely-many, but countable). Having those abstractions allows us to give a finite interpretation over which our proof system is based. Due to the properties of our models, we however have to depart quite largely from the axiomations using symbolic bisimulations proposed in [Lin94] and [LY00].

In Chapter 5 we provide a behavioral type system for the language defined in Chapter 3. This type system allows, for a subset of potentially infinite-state configurations, to check that no error occurs due to a contract violation. The method is compositionnal by essence, using some kind of *assume/guarantee* reasoning: for an open system, a well-typing result states that no communication error may occur (guarantee) under the condition that any environment provided for it is well-typed as well (assumption). In that case, a type corresponds to a deterministic specification of the interface between a component and its environment. A notion of *subtyping* is also defined, allowing the *substitution principle* to be applied: any component of type A may be replaced by another component of type B provided that A is a subtype of B, noted $A \preceq B$. We hence obtain a top-down way of building safe configurations by successive refinements.

In Chapter 6 we first present a short overview about the notion of contract, as it can be found in existing proof systems and specification laguages. We then propose two applications for our development technique. The first proposal promotes the use of π^δ in real-time middleware, for monitoring, security, and evaluation purposes. The second second proposal is directed towards the less formal but more widespread UML-RT notation for real-time software engineering. We show the concepts defined in the theory for π^δ can be used advantageously to encode UML-RT specifications. We then take an example of such conception using π^δ.

The final chapter is devoted to our conclusions.

Chapter 2

Open Distributed Processing and Real-Time

2.1 Open Distributed Processing

2.1.1 Overview

The goal of the *Open Distributed Processing* standard [ODP95] has been, since its introduction in 1995, to provide a reference model precisely identifying which concepts and relations should be used during the development of distributed applications. In particular, RM-ODP promotes multi-aspect specification, object encapsulation, model composition/decomposition, and model abstraction/refinement. RM-ODP is conversely not tied to any concrete notation or modeling language, and it is the responsibility of the application designer to define or select languages for system description where RM-ODP concepts can be represented the most easily.

ODP introduces five viewpoints from which the development of a given application can be considered. Each viewpoint is suited for the expression of a particular aspect of the application. However, a given property may have several aspects, concerning many viewpoints: viewpoints may overlap (and they usually do). ODP hence fosters the use of formal method to help establishing viewpoint consistency. The following viewpoints definitions are given in the first part of the standard [ODP95].

- The *enterprise* viewpoint: a viewpoint on the system and its environment that focuses on the goals and policies retained for the system.
- The *information* viewpoint: a viewpoint on the system and its environment that focuses on information significance and information processing issues.
- The *computational* viewpoint: a viewpoint on the system and its environment that enables the expression of distribution-related issues by attaching functionalities of the system to distributed computational entities called *objects*.
- The *engineering* viewpoint: a viewpoint on the system and its environment that focuses on the mechanisms and functions required to support distributed interaction between objects in the system.

- The *technology* viewpoint: a viewpoint on the system and its environment that focuses on the choices of technologies for that system.

2.1.2 The Computational Viewpoint

Our interests will focus on the computational viewpoint. In that viewpoint, the application is modeled as an evolving set of objects having interfaces that they use to interact synchronously or asynchronously with other objects over supposedly perfect or lossy channels.

2.1.2.1 Objects and Services

Objects are constituted of an arbitrary behavior and a finite encapsulated data set. Concrete computational modeling languages have to respect two principles in order to be ODP-compliant. The *object abstraction* principle says the programmer may ignore irrelevant details: an object may be the result of the composition of other objects, an object may have any granularity (*e.g.* from a simple integer to a complex factory system), and the services offered by an object may be described in a way independant of its implementation. The *data encapsulation* principle oppositely restrains object behavior: the only way an object B has to produce a side effect on the data of an object A is by calling a method of A.

An object may provide or access services through its *interfaces*. An interface is a set of available methods together with conditions indicating when each method may be invoked. An interface will be said to play a *role* in the provision of some service. Two examples of roles are the *server* that provides a service, and the *client* that accesses it. Services (and interfaces) come in two flavors: flow services and operational services.

ODP indeed distinguishes two forms of object interactions. *Flow* services are intended to abstractly represent *continous* interactions: the *source* role of a flow service sends a stream of data to the *target* role of that service. *Operational* services oppositely discern each interaction as an individual event: an operation call occurs between a *client* and a *server* role, resulting in a method invocation on the object owning the server role. An operation can be either an *interrogation*, where the server returns a response, or an *announcement*, where no response is sent to the client. An example configuration of objects communicating through operational interfaces is presented on Figure 2.1.

The object on the left possesses an interface holding the client role of a service u. Its invocations always concerns the server interface for u that is owned by the object on the right. *Signals* for operation invocation are conveyed by *messages*, and on our example a message carrying signal m circulates from the client to the server. If m is an interrogation operation, then a reply signal also named m will be sent later by the server. It can be seen that roles and interfaces may confound easily, and we will not restrict from making this confusion, often using "role r" for "interface playing role r"; should the meaning not be clear from context, we shall always give necessary precisions.

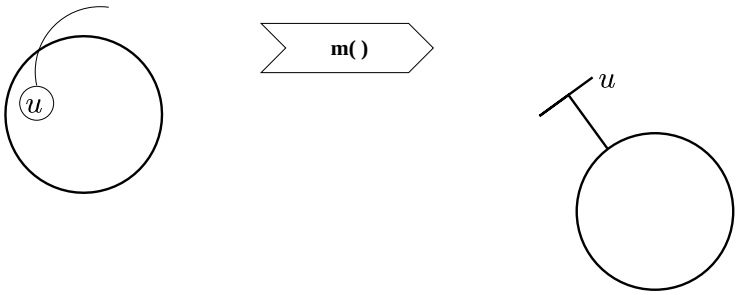

Figure 2.1: Communicating roles of an Operational Service u

2.1.2.2 Passing Roles Around

We now show how a given configuration may dynamically evolve by passing around the roles it possesses. A message may indeed be sent with roles as effective parameters, as channels can be passed around in formalisms such as the π-calculus. An example illustrating this feature is depicted in Figures 2.2 and 2.3.

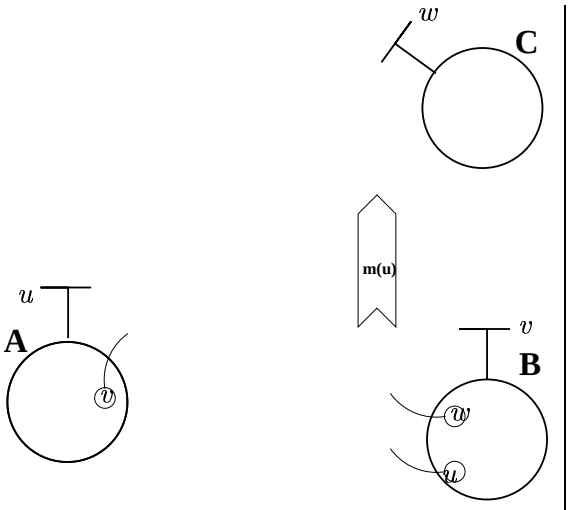

Figure 2.2: An example of communicating interfaces, step (1)

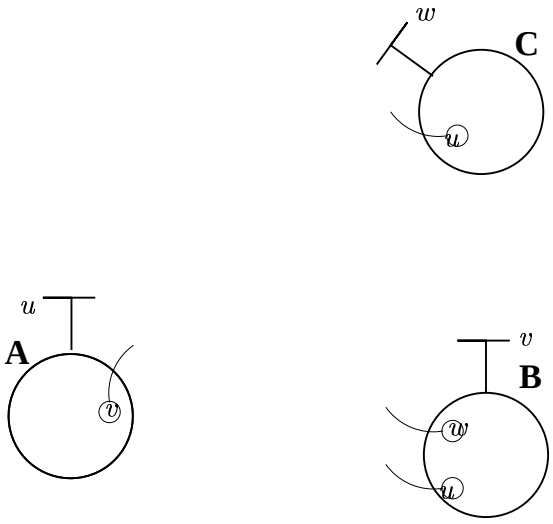

Figure 2.3: An example of communicating interfaces, step (2)

The message m shown on Figure 2.2 has been sent by object **B**. It features a copy of the client role of service u as effective parameter, and it is received by object **C** on Figure 2.3. The object **C** now has the capability to send messages to the owner **A** of the server role of service u. As so does object **B** (it did not loose its client role on u but sent a copy of it), both **B** and **C** can concurrently access service u provided by object **A**.

2.1.2.3 Contracts and Object Binding

When reasoning about services and object interactions, a fundamental point is the possibility to explicitly declare, negociate, observe and enforce applicative Quality of Service (QoS) properties. RM-ODP tackles this issue by introducing the pervasive

notion of *contract* among entities. A contract is a written document enclosing the nun-functionnal properties sustained by a service provider or required by a service accessor. The establishment of a service session is generally called *binding*. In a binding the accessor and provider objects are bound under the rules of some contract. A contract may itself include rules declaring how object binding must be done in order to properly establish the corresponding service. RM-ODP then separates two cases. If the contract imposes that any object willing to become client of a service must (directly or indirectly) contact the provider *before* establishing the service, then the binding procedure is said *explicit*. A *binding object* may then be created upon service establishment to manage the connection: if the contract is *broken* by some *faulty* object involved in it, the binding object may either shut the service or take any necessary measure in conformance with the contract terms. On the other side if any object can access a service without pledging allegiance to the server beforehand the binding is said *implicit* and no binding object may be created.

It is often that the writing of contracts actually becomes unnecessary when some *contract framework* is used. A contract framework contains service-kind specific molds for contracts: when applied to a service of some kind, a mold produces the contract ruling that service. A contract mold describes how the service must be established (*i.e.* binding rules) and under which conditions the contract can be maintained. The latter rules imply a definition of *role compatibility* (*i.e.* peered roles are well-behaved regarding each other), a notion that leads to *contract consistency* (*i.e.* a contract is consistent iff its roles are compatible). Under a *contract framework*, the notions of *service* and *contract* may therefore be united without further concern.

In conclusion we underline that having formally-defined contracts may clearly allow for both modular specifications (they can be compared to the abstract data types of sequential programming) and compositionnal verification.

2.1.3 The Engineering Viewpoint

We will refer to the engineering viewpoint in the application part of this thesis. Yet we now provide a short overview of it.

At engineering level, objects are not distribution-aware anymore. Instead, they are considered according to the architecture which Figure 2.1.3 is an example. A node is the physical unit for distribution. It contains capsules, which are the unit for processing and failure (eg UNIX processes). Within capsules, highly-interacting objects are then grouped into *clusters*. Each node has a *nucleus*, that provides a service to the capsules of its node through an interface called the *Node Management Interface* (NMI). Each capsule has a *capsule manager* object that provides a service to the objects of its capsule through the interface similarly called the *Object Management Interface* (OMI). The nucleus can be seen as the operating system kernel of the node, whereas capsule managers can be seen as virtual machines executing clusters of objects.

Objects within a cluster are linked directly by their interfaces, whereas other objects must use *channels*. Channels encompass stub objects (for marshalling, etc.), binder objects (to administrate the link), and protocol objects (for communications).

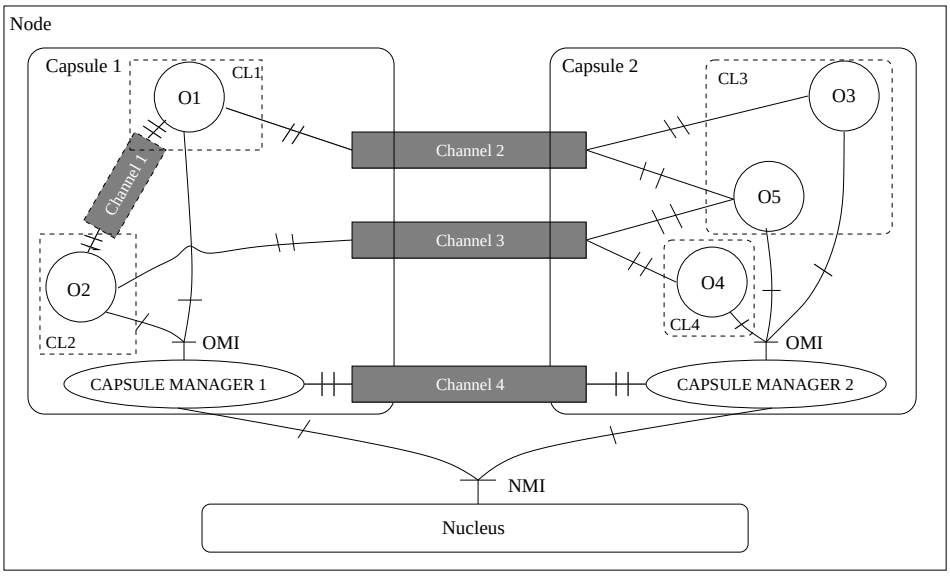

Figure 2.4: An ODP Configuration at Engineering Level

2.1.4 Specifications, Viewpoints and their Relations

As suggested by [BS97], engineering and computational viewpoint specifications should use the same language. Following this proposition, we use $\mathcal{A}$rtOC along those different activities. Hence, the *Virtual Machine* (VM) specification is given at engineering level by the *system architect*, while user objects are viewed at computational level by the *application designer*. Each objet from a given user application is subsequently mapped to one at engineering level.

It should however remain to the mind of the reader that the semantics of $\mathcal{A}$rtOC is essentially different at those two levels: a computational view implies maximum parallelism, whereas engineering view implies concurrency through interleaving within each capsule. The goal of the VM architecture and policies is to preserve the computational level semantics at engineering level: they can be considered equivalent if any required resource at computational level is granted at engineering level.

The body of this thesis consists in the presentation of a language and verification methods staged at the computational level. As an application, a short description of what could be a VM API at the computational level, as well a description of its inner components at engineering level, will be given in Chapter 7.

2.2 Modeling and Verifying Real-time Systems

It is not possible for us to make an exhaustive summary of the developments accomplished in formal specification dealing with real-time over the last thirty years. Let us just write that, as methodologies and languages have endured a tremendous explosion in number, adding time could never have been perceived as a way of simplifying the encountered problems.

In a nutshell, we adopt a solution relying on a *dense* notion of time, while concurrency is reduced by *interleaving*, and we devise a *compositional* static analysis

method that considers *non-probabilistic* models and evaluations.

Having a dense time domain is opposed to having a *discrete* time domain [RT91]. We adopt a dense time domain because it is more "abstract" in nature [R. 91], and therefore allows to refine and compose specifications more easily. Discrete time is present in many formalisms, such as for example synchronous languages like Esterel [BG92] and Lustre [HLR92], and several process algebras [NS92]. Chosing a discrete notion of time has also been advocated as a simple method that can be used to extend time-insensitive tools with timing features [CCMM95].

However, the dense notion of time have widely spread, since the advent of Alur and Dill's *timed automata* [AD94]. Indeed, an inherent problem of having a dense time domain is that models have infinitely (even uncountably) many states, making their analysis difficult if not impossible. However, Alur and Dill introduced in [AD94] an abstraction method allowing to perform an exact verification on a discretization of the dense state space; the result of this conservative abstraction is called the *region graph*. From there, many results about decidability and undecidability of essential problems have been proved using the region graph.

An obvious way to obtain a language with real-time features is to extend an untimed language. The extension can be conservative or not: properties of untimed specifications can easily be lost when time is introduced. An important point is to aboid such discrepancies. We proceed in this way with the π-calculus. Before us, we can cite linear-time or branching-time temporal logics (TCTL [ACD93], TPTL [AH89], MITL [AFH91]), process algebras [Yi91, Mol90, BB91, NSY93, Sch95, CdO95, LL92, Sig99, NS92]. Even there, many variations exist in the models produced in those formalisms [RT91].

We favor an operational approach to semantics, resolving concurrency to non-determinism (see, for example [Mil89a]). Other proposals adopt *true concurrency* semantics, providing a more denotational feel: [Fu95, AM96]. In that category can also be classified the various extensions brough to petri nets, such as [Ram74, MF76]. We did not explore that way of doing things, preferring to go in unison with the large majority of π-calculi litterae.

Chapter 3

An Untyped π^δ: Syntax and Semantics

3.1 Introduction

The π-calculus [MPW92, SW01] is today a widely accepted formalism for the specification and the verification of open distributed systems, where the interaction topology among the components of an application may dynamically evolve during the execution. However, many open applications also exhibit timeliness requirements, that the π-calculus is unable to represent. Important examples include multimedia systems [CBS+95], active networks [WJGO98] and QoS-enforcing middleware [Sch00].

However, all the widely recognized timed process algebras assume purely *static* interaction topologies [Yi91, Mol90, BB91, NSY93, CdO95, NS92]. They are extensions of CCS [Mil89a], CSP[Hoa85], Lotos [BB89] or ACP [BW90, BK84] in either a discrete or a dense time setting. There is only one proposition of a π-calculus with discrete timers [BH00], which is application-directed and gives no algebraic properties such as axiomatization or expansion laws. We propose a timed π-calculus featuring *clocks* valued over a dense time domain, and discuss some of its algebraic properties.

We consider three primary goals:

- our calculus must conservatively extend the algebraic theory of the π-calculus,
- a process should be able to perform actions according to its own will, hereby assuming a certain *autonomy* towards its environment, and,
- the calculus must allow a certain form of compositional reasoning and modular verification.

To satisfy the first condition, we give a meaning to all the (untimed) π-calculus processes in our timed calculus: any process that does not specify any time-related property is deemed to be *patient*, meaning that it may let time pass without restriction. All patient processes have therefore indistinguishable timely behaviors. This effective solution for embedding an untimed semantics into a timed one has been

suggested by many authors, from Hennessy and Regan [HR95], to Lynch and Vaandrager (using so-called *patient transducers*) [VL92]. It results that, for a large body of calculi, the algebraic laws for untimed processes may then be applied straightforwardly to the timed calculus. We shall study the relevance of this assertion in a π-calculus setting.

Unfortunately, the second and third goals fail to retain such simple solutions, deserving a much more subtle treatment. We now address process autonomy, for which our calculus provides an original solution; compositionality will be the object of the next chapter.

Process autonomy has been identified since the late 80's as a major concern for distributed system designers. The question is: should the environment of a process be able to block the process' actions? The answer by Lynch and Tuttle [LT87], who were to our knowledge the firsts to state the question clearly, was: *never*. Before that, the designers of early process algebras (such as CSP or CCS) implicitly replied: *always*. Since then, authors of timed process algebras have also answered: *sometimes*. We cheerfully adopt the third solution. In the next two paragraphs however, we shall briefly expose and relate those three points of views on process interaction.

Input/Output automata were proposed by Lynch and Tuttle [LT87]. Those automata are able to synchronize on actions, each automaton having an *interface* that describes which actions are under its control (*output* actions) or under the control of its environment (*input* actions). Any automaton must be *input enabled*: it has to always be ready to synchronize on any of its declared input actions. Furthermore, two automata can not be composed in parallel if the intersection of their output sets is not empty. This guarantees that an automaton is the one that is able to perform the output actions present in its interface, and that its environment cannot refuse to synchronize on them.

An essential argument for the development of I/O automata was the fact that a CSP process can be impeached to perform an action by placing it in an environment that refuses this action forever. Conversely, a process may deceptively appear to provide a good solution to a problem in some environments (the ones that refuse the "bad" actions) while behaving inappropriately in other ones. Process algebraists replied by invoking the classical *bounded buffer* example. A n-place buffer may perform a *put* action only when it is not full, and a *get* action only when it is not empty. Making a buffer responsive to those signals at any time shows only meager pertinence.

Our needs lie precisely in-between those two worlds; we intend our components to clearly state what are their requirements, and what they either wish, accept, or tolerate (indistinctly). We henceforth take advantage of having time-related executions: timed processes may apply *selection* and *urgency* conditions on action prefixes. Selection indicates when an action *may* occur (at any other point in time the action is precluded), while urgency indicates when an action *must* occur (if the environment refuses to interact then the whole system produces an error). Almost all other timed process algebras feature urgent actions, although in rather different settings; see Section 3.9 for a detailed comparison. We figure that the timed process algebra that is the closest to ours is the *urgent Lotos* (U-Lotos) of Bolognesi and

Lucidi [BL92b, BL92a]. In U-Lotos, the operator $\texttt{asap}_{\texttt{a}}$ allows to force urgency on the next occurrence of an action from the given set `a`. Yet, our introduction of name mobility in this chapter, the subsequent formal treatment that we provide in the next, and the compositionality issues that we address in Chapter 5 are new, at least to our knowledge.

The remainder of this chapter is organized as follows. In the next section, we present the syntax of our timed π-calculus. We then provide a semantic for it based on timed labeled transition systems. After what we give a small example and compare the time-related properties of our models to the ones of models yielded by household real-time process algebras. Finally, we further discuss related works and conclude in Section 3.9.

3.2 The Syntax: Processes, Ports, and Clocks

The simplest elements of the π-calculus are *names*. In the untimed π-calculus, names can be used as communication devices, called equivalently *ports*, *gates*, or *channels*. We add a second sort of names, called *clocks*. We hence consider two recursively enumerable sets, $\mathbf{P}$ for ports and $\mathbf{C}$ for clocks. Typical ports are named a, b, c. Typical clocks are named x, y, z. We shall denote the usual definitional equation using the $\triangleq$ sign; we name $\mathbf{N} \triangleq \mathbf{P} \cup \mathbf{C}$ the set of all names, which typical elements are m, n, o. Any name can be adjuncted the usual decorations, such as primes, indices and exponents.

The syntax for a process prefixing action π is given in two steps by

$$\pi^\star ::= a(n) \mid \overline{a}\, n \quad \text{and} \quad \pi ::= \tau \mid \pi^\star \,.$$

The silent action τ can be thought of as an internal action of the process. Any other prefix has a *subject* a in $\mathbf{P}$ and an *object* n in $\mathbf{N}$. Informally, $\overline{a}\, n$ stands for the *output* of object n over subject a, while $a(n)$ stands for the *input* of name n over a. For any name a, actions with respective subjects a and $\overline{a}$ are said *complementary*. The subject is said to be *free* in any prefix, whereas the object is *bound* in $a(n)$ and free in $\overline{a}\, n$. *Process terms*, typically ranged over by t, u, v and their decorated variants, can be built using the following grammar in Backus-Naur Form:

$$t ::= 0 \mid \mathit{Error}^{\upsilon} \mid [\sigma,\, \upsilon]\pi.\, t \mid [a = b]t \mid \nu n\ t \mid t + t \mid t|t \mid A(\tilde{n}) \,.$$

The constraint σ and the urgency condition υ (a.k.a. the *deadline*) respect the following syntactic forms, given respectively by

$$\sigma ::= \sigma \wedge \sigma \mid x \sharp^{\sigma} q \mid x - y \sharp q \quad \text{and} \quad \upsilon ::= \upsilon \wedge \upsilon \mid x \sharp^{\upsilon} q \mid x - y \sharp q$$

where $\sharp \in \{<, \leq, \geq, >\}$, $\sharp^{\sigma} \in \{<, \leq, \geq\}$, $\sharp^{\upsilon} \in \{<, \geq\}$, and q is a time value. The process $\mathit{Error}^{\top}$, where $\upsilon = \top$ is the ever-true condition, is noted Error for short.

Processes with term 0 are called *idle* processes, while processes with term Error are *deadlocked* processes (we also prosaically call them *error* processes). Process operators are named, we list them as they appear in t from the left to the right: prefixing, matching, restriction, sum (a.k.a. choice), composition (a.k.a. parallel),

and process instantiation. Operator precedence is as follows: prefixing, matching and restriction bind more tightly than sum, and sum binds more tightly than composition. All binary operators associate to the left. We shall use parentheses freely to oppose or precise precedence rules.

Each occurrence of a name n in the subterm t of a term $(n)t$ or $a(n).t$ is a *bound* occurrence. We define $\mathrm{bp}(t)$ and $\mathrm{bc}(t)$ as respectively the set of bound ports and the set of bound clocks of a term t; the set of *bound names* of t is $\mathrm{bn}(t) \triangleq \mathrm{bp}(t) \cup \mathrm{bc}(t)$. Any occurrence of a name in a term t that is not bound is said *free*; we name $\mathrm{fp}(t)$, $\mathrm{fc}(t)$ and $\mathrm{fn}(t)$ the sets of free ports, free clocks, and free names of t. The set of names of a term t is $\mathrm{n}(t) \triangleq \mathrm{fn}(t) \cup \mathrm{bn}(t)$. We allow the generalization of those functions to prefixes π, using the definition of bound and free prefix names given above.

A typical list of names $n_1, n_2, \cdots, n_k$ will be noted $\tilde{n}$, its cardinal being $|\tilde{n}| \triangleq k$. The corresponding (unordered) set containing each n_i comprised in $\tilde{n}$ will be noted $set(\tilde{n}) \triangleq \{n \mid \exists i.\ 1 \leq i \leq k \wedge n = n_i\}$. The *simultaneous substitutions* in a prefix π or in a term t of the free occurrences of each name m_i from a list $\tilde{m}$ by the corresponding name n_i from a list $\tilde{n}$ (with $|\tilde{n}| = |\tilde{m}|$) are noted respectively $\pi[\tilde{n}/\tilde{m}]$ and $t[\tilde{n}/\tilde{m}]$. The substitution may as usual involve the renaming of bound names to avoid incidental capture of free names. We range over substitutions with α, β, γ and their decorated variants. Substitution has the highest precedent over all process term operators. In general, terms are considered equivalent up to α-conversion of port names.

A *process* conjoins a process term t with a contextual *clock valuation* ρ, which is a partial function $\rho : \mathbf{C} \rightharpoonup \mathbb{T}$ yielding a value in the time domain $\mathbb{T}$ for each clock name appearing free in t. We denote this conjoining by valuation post-fixing, an operation that has a lower priority than all process operators: $t|u\rho$ is equal to $(t|u)\rho$; we will often enforce this fact by using explicit parentheses, though. Finally, fixing a last vocabulary issue, in a non-error process $t\rho$ we will often designate the term t as being the *control part* of the process, and ρ to be the *context part* of the process.

For some process $t\rho$ and some clock $x \in dom(\rho)$, we name *ceiling for x in t* and note $ceil(x, t)$ the highest constant appearing in the term t within a selection or urgency condition constraining x.

The semantics for our processes alternates *time-passing* actions and *discrete* actions: silent, input and output actions take no time to execute[1]. We now give an informal explanation of the meaning of each process $t\rho$. A process with 0 as control part may only let time pass, without bound. A process with term $Error^{\upsilon}$ can let time pass until ρ satisfies υ; from this time it can not do anything at all. Such deadlocked processes play a distinctive role in our theory: they are to be reached for example when a process requiring some interaction is not given satisfaction by its environment. Process $[a = b]t\rho$ executes $t\rho$ whenever a is syntactically equal to b, or behaves as the idle process otherwise. Process $(\nu n\ t)\rho$ adopts the actions of $t\rho$ with n hidden if n is a port name, or it extends the valuation ρ to associate the value 0 to n if n is a clock name. In both cases, the continuation t is executed immediately in the new context. Process $(t + u)\rho$ represents the choice between t and u, allowing time to progress if both t and u agree, or executing the first process being

[1] we can classify our semantics as *point-based* and *weakly monotonic* according to [RT91].

able to synchronize with the environment, abandoning the other. Process $(t \mid u)\rho$ executes t and u in parallel within context ρ. For process terms $A(\tilde{n})$, we assume a set of process names (typically ranging over $A, B, C \ldots$), each associated to a list of formal parameters and a process term which free names are included in the formal parameter list. Then $A(\tilde{n})\rho$ executes the term associated to A in context ρ, having orderly replaced the formal parameters in the term by the effective arguments $\tilde{n} = n_1, \cdots, n_k$. Such an equational definition for a name A is noted as we do in other occasions, $A(\tilde{n}) \triangleq t$, with $\mathrm{fn}(t) \subseteq \mathit{set}(\tilde{n})$.

Last but not least, we give to processes of the form $([\sigma, \upsilon]\pi.\, t)\rho$ the particular attention they deserve. Basically, those processes refuse synchronization on π when ρ makes σ false, consent to it when ρ makes $\sigma \wedge (\neg\upsilon)$ true, and require it when ρ makes υ true (σ is then true, since we impose $\upsilon \Rightarrow \sigma$). When such a process requires synchronization on π, time may pass as long as the environment does not offer (or requires) synchronization on π. If the environment refuses the synchronization and υ is true, the process may allow υ to become false again under the condition that the whole configuration reduces to *Error*. If the process does not require the synchronization but only consents to it (υ is false), then time may pass freely, until either σ becomes false (synchronization is by then refused), or υ becomes true (synchronization is by then forced); the synchronization may happen if it is consented to by the environment, or required by it. If the process refuses to synchronize, then the environment may by symmetry lead the whole configuration to reduce to *Error* by requiring synchronization to happen.

3.3 The Time Domain and Its Properties

Before we can give a semantics for processes, it is necessary to fix a time domain for clocks and to explicitly state what are its properties. We name the time domain $\mathbb{T}$, and its typical elements the diversely decorated forms of δ. For time domain we can take any commutative monoid over some partially ordered set with unit 0 and addition as binary operation. We impose $\mathbb{T}$ to have the two following properties :

- *left-cancellative*: $\delta_1 + \delta_2 = \delta_1 + \delta_3 \;\Rightarrow\; \delta_2 = \delta_3$, and
- *anti-symmetric*: $\delta_1 + \delta_2 = 0 \;\Rightarrow\; \delta_1 = \delta_2 = 0$.

We then define a *precedence relation* over $\mathbb{T}$: $\forall\delta_1, \delta_2.\, \delta_1 \leq \delta_2 \;\Leftrightarrow\; \exists\delta_3.\, \delta_1 + \delta_3 = \delta_2$. From this we can easily deduce that 0 is the *least element* of $\mathbb{T}$. The *strict precedence* relation is defined by: $\delta_1 < \delta_2 \Leftrightarrow \delta_1 \leq \delta_2 \wedge \delta_1 \neq \delta_2$. It has been observed by most authors of time-enabled theories that having a *dense* time domain is extremely valuable during specification activities: it provides a clean and abstract time scale that can be refined at will. Formally, density is defined by: $\forall\delta_1, \delta_3.\, \exists\delta_2.\, \delta_1 < \delta_2 < \delta_3$. Adopting this *modus operandi*, we assume that $\mathbb{T}$ is dense; we even become more specific and, also with many authors, we adopt the set of positive real numbers $\mathbb{R}_+$ for time domain: $\mathbb{T} = \mathbb{R}_+$.

3.4 Clock Valuations, Constraints, And Satisfaction

We first have to define several operations on clock valuations, that will be used in the operational semantic rules. We first define *valuation juxtaposition*, assembling valuations ρ and κ into valuation $\rho\kappa$. Juxtaposition is defined if either $dom(\rho) \cap dom(\kappa) = \emptyset$, or ρ and κ agree on the values of the clocks for which they are both defined: $\forall x, \delta, \delta'.\ (\rho(x) = \delta \ \wedge \ \kappa(x) = \delta') \Rightarrow \delta = \delta'$. When defined, $\rho\kappa$ has $dom(\rho) \cup dom(\kappa)$ for domain, and $(\rho\kappa)(x) \triangleq \rho(x)$ if $x \in dom(\rho)$ and $(\rho\kappa)(x) \triangleq \kappa(x)$ if $x \in dom(\kappa)$.

We define the substitution operator on clock valuations ρ by: $dom(\rho[\tilde{x}/\tilde{y}]) \triangleq ((dom(\rho) \backslash set(\tilde{y})) \cup set(\tilde{x}))$ and, for any i with x_i in $\tilde{x}$, $(\rho[\tilde{x}/\tilde{y}])(x_i) \triangleq \rho(y_i)$. Meanwhile, $(\rho[\tilde{x}/\tilde{y}])(z) \triangleq \rho(z)$ for any $z \notin set(\tilde{x})$.

The clock-resetting operator $\rho^{\downarrow x}$ is defined only when $x \notin dom(\rho)$. It then yields a valuation identical to ρ except that its domain comprises the clock x, the value assigned to which being 0, verifying: $\rho^{\downarrow x}(x) = 0 \wedge (\forall y, \delta.\ \rho(y) = \delta \Rightarrow \rho^{\downarrow x}(y) = \delta)$.

The valuation $\rho^{+\delta}$ has the same domain as ρ but yields a value superior by δ for all the clocks in its domain: $(\forall x, \delta.\ \rho(x) = \delta \Rightarrow \rho^{+\delta}(x) = \rho(x) + \delta)$. The restriction operator $\rho^{\backslash \tilde{m}}$ yields an environment identical to ρ but undefined for the clocks in $\tilde{m}$:

$$(dom(\rho^{\backslash \tilde{m}}) = dom(\rho) \setminus set(\tilde{m})) \wedge (\forall x, \delta.\ (x \notin set(\tilde{m}) \wedge \rho(x) = \delta) \Rightarrow \rho^{\backslash \tilde{m}}(x) = \delta)\ ;$$

it trivially yields ρ if $\tilde{m}$ contains only port names. The converse operator $\rho^{/\tilde{m}}$ is the restriction of ρ to clocks in $\tilde{m}$:

$$(dom(\rho^{/\tilde{m}}) = dom(\rho) \cap set(\tilde{m})) \wedge (\forall x, \delta.\ (x \in set(\tilde{m}) \wedge \rho(x) = \delta) \Rightarrow \rho^{\backslash \tilde{m}}(x) = \delta)\ .$$

We shall assume that all those operators have the same precedence, and that they associate from left to right, yielding for example: $\rho^{\downarrow x+\delta} = (\rho^{\downarrow x})^{+\delta}$.

We finally define the equality relation over two valuations ρ and κ as true if and only if $dom(\rho) = dom(\kappa)$ and for any $x \in dom(\rho)$ we have $\rho(x) = \kappa(x)$.

We now define constraint satisfaction for a family of constraints ζ, which are more general than the constraints σ and υ used in process terms; we will need this definition later, and it is downward-compatible. So, consider

$$\zeta ::= \zeta \wedge \zeta \mid \zeta \vee \zeta \mid \neg\, \zeta \mid x \sharp q \mid x - y \sharp q$$

where $\sharp \in \{<, =, >\}$ and $q \in \mathbb{Q}_+$. We shall sometimes use the abbreviation $x \leq q$ for $(x < q \vee x = q)$ and $x \geq q$ for $(x > q \vee x = q)$, respectively. The $true$ constraint, noted $\top$, is defined by $(x - y < q \vee x - y \geq q)$ for some clocks x, y and some constant q. The $false$ constraint, noted $\bot$, is defined by $(x - y < q \wedge x - y \geq q)$ also for some clocks x, y and some constant q. We note $clocks(\zeta)$ the set of all clock names appearing is ζ. By analogy with process terms, $\zeta[\tilde{x}/\tilde{y}]$ denotes the constraint obtained after the simultaneous substitution, for each clock name x_i, of all the occurrences of x_i in ζ by y_i.

We denote the satisfaction of a constraint ζ by a clock valuation ρ as: $\rho \vDash \zeta$. It

is defined by:

$$\begin{array}{lll} \rho \vDash x \sharp q & \Leftrightarrow & \rho(x) \sharp q \\ \rho \vDash x - y \sharp q & \Leftrightarrow & \rho(x) - \rho(y) \sharp q \\ \rho \vDash \zeta \wedge \zeta' & \Leftrightarrow & \rho \vDash \zeta \wedge \rho \vDash \zeta' \\ \rho \vDash \zeta \vee \zeta' & \Leftrightarrow & \rho \vDash \zeta \vee \rho \vDash \zeta' \\ \rho \vDash \neg\, \zeta & \Leftrightarrow & \neg\, (\rho \vDash \zeta) \end{array}$$

Of course, for $\rho \vDash \zeta$ to be defined, we must have $\mathit{clocks}(\zeta) \subseteq \mathit{dom}(\rho)$. When satisfaction is defined but fails, we may also note $\rho \nvDash \zeta$.

The semantics of a constraint is given as $[\![\zeta]\!] \triangleq \{\rho \mid \rho \vDash \zeta\}$, the set of clock valuations satisfying ζ. We then have $[\![\bot]\!] = \emptyset$, $[\![\top]\!] = (\mathbf{C} \rightharpoonup \overline{\mathbb{R}_+})^n$ for any integer n, $[\![\zeta_1 \wedge \zeta_2]\!] = [\![\zeta_1]\!] \cap [\![\zeta_2]\!]$, $[\![\zeta_1 \vee \zeta_2]\!] = [\![\zeta_1]\!] \cup [\![\zeta_2]\!]$, and $[\![\neg\, \zeta]\!] = \overline{[\![\zeta]\!]}$. For general clock constraints ζ_1 and ζ_2, we define $\zeta_1 \backslash \zeta_2 = \zeta_1 \wedge \neg\, \zeta_2$, and $\zeta_1 \Rightarrow \zeta_2$ logically as $\neg\, \zeta_1 \vee \zeta_2$. We obtain trivially $[\![\zeta_1 \backslash \zeta_2]\!] = [\![\zeta_1]\!] \cap \overline{[\![\zeta_2]\!]} = [\![\zeta_1]\!] \backslash [\![\zeta_2]\!]$ and $[\![\zeta_1 \Rightarrow \zeta_2]\!] = \overline{[\![\zeta_1]\!]} \cup [\![\zeta_2]\!]$.

For constraints appearing in process terms, since we miss negation and therefore cannot define implication logically, we define implication semantically: $\vDash \sigma \Rightarrow \sigma'$ if and only if $\forall \rho.\ \rho \vDash \sigma \Rightarrow \rho \vDash \sigma'$ (or equivalently $\forall \rho.\ \rho \in [\![\sigma]\!] \Rightarrow \rho \in [\![\sigma']\!]$).

In both cases however, non-implication, equivalence, and non-equivalence follow straightforwardly from the respective logic and semantic definitions for implication.

3.5 Well-Formedness Criteria for Processes

The first assumption we make is that time constants appearing in process terms are natural numbers, although time ranges on positive reals. Actually, process terms may also use rational numbers, which yield no further complications. The limitation to rational or natural constants does not hamper the expressiveness of our formalism, since any term featuring rational constants has an isomorphic term featuring only natural constants: the notion of time we use is abstract enough so that a property verified on some process is true on all processes for which all time constants have been multiplied by a fixed number. This naturally corresponds to constricting or expanding the time scale.

The use of rationals (or natural numbers) in constraints is required to apply abstraction methods such as the *region graph* construction (see Section 4.1). We will need them to propose in this chapter a proof system, and in the next a decidable type system.

We shall only give a semantics to *well-formed* processes, imposing statically decidable constraints on them. First, for any process $t\rho$, all the free clocks in t should be given a value by ρ: $\mathrm{fc}(t) \subseteq \mathit{dom}(\rho)$. We also impose equations of the form $A(\tilde{n}) \triangleq t$ to feature only *well-guarded recursion*: calls to A in t should only occur after a prefix action, hence being of the form $[\sigma, \upsilon]\pi.\, A(\tilde{n})$.

We also impose that for any action π the urgency condition implies the selection condition: $\vDash \upsilon \Rightarrow \sigma$, a process being able to impose urgency on a prefix action only if this action is available to its environment. In this, and also in the limitation to left-closed clock conditions (this is syntactic, from the definition of υ and σ in Section 3.2), we follow [BST97]. We however further restrain urgency conditions to

be right-open: we have to be able to know the exact time when an error occurs, since time progression may not continue further past this exact time.

3.6 The Semantics: Labeled Transition Systems for Processes

As usual in the process algebra community, We define the semantics of π^δ terms as a Timed (labeled) Transition System (TTS). Our semantics therefore associates a TTS to each well-formed process. Semantic rules are given in the Structural Operational Semantics (SOS) style [Plo81].

As in existing timed process algebras and most π-calculus theories, we opt for an *interleaving* semantic treatment of concurrency [Yi91, Che92]. Each transition is labeled with only one action, meaning that either one process has evolved on his own and other processes have waited, or that many processes have agreed to collaborate in performing this action. The complementary approach, named *true concurrency*, has also fostered a large body of work; an extensional survey and comparison of both approaches can be found in [WN95]. Truly concurrent models for the π-calculus can be found in [San96, DP99].

The semantics of our processes yet exhibit several differences with other interleaving semantics found in the literature. Indeed, we think that whenever a process forces the occurrence of an action by applying an urgency condition to it, its apparent behavior is different than the behavior of a process proposing the same action without urgency condition. Rephrased, this is equivalent to say that selection and urgency conditions should be part of the observable behavior of a process. In apparent contradiction, we also think that when an interaction occurs, it is not relevant to know if this interaction was forced by one of the participants or not. What matters is that the interaction *did occur*. In our view, selection and urgency hence do not have the greatest influence on interactions, but on time progression. Even more, we think it easily appears that selection and urgency do not merely modify but entirely arbitrate time progression. This arbitration is two-folded: it encompasses as well prohibiting time to pass when two processes are able to interact and one of them imposes urgency on the interaction, as producing an error when a process imposes a deadline on an interaction and its environment is willing to let time go passed this deadline.

Hence, we think a process should not only state "I allow this delay to pass", but: "I allow this delay to pass, meanwhile I wish to interact on these ports as soon as possible and I offer synchronization on these other ports". Outside of the inner interest of the model, we remark that this model seems the least reasonable one allowing urgency conditions to be effectively enforced while the semantics is defined by structural induction over the structure of terms (*i.e.* we obtain a *compositional* semantics in the sense of [Plo81]). To our knowledge, in this respect, no comparable model can be found in the literature (we elaborate a little more on this point in Section 3.9).

The labels of our transitions therefore respect the syntax:

$$\lambda ::= \pi \mid \overline{a}(b) \mid \delta\, \mathcal{S}$$

where the delay δ chosen among the strictly positive real number $\delta \in \mathbb{R}_+^*$, and its adjoined *interaction ready set* $\mathcal{S}$ adopt the form $\langle \mathcal{R} : \mathcal{O} \rangle$, $\mathcal{R}$ being the prefix set of interaction *requests*, and $\mathcal{O}$ the prefix set of interaction *offers*. The prefix sets indicate, for each interaction:

- the name of the subject,
- the sort of the object (port or clock).

Typical members of a prefix set are thus written a^c, a^p or $\overline{a}^c$, $\overline{a}^p$. What is *not* distinguished by offer and requirement sets is the number of offers having the same subject and the same object sort, and for each prefix the precise value of the object that is to be transmitted as well as whether this object is bound or free.

We shall use only one binary operation on ready sets, which is the disjoint union of their request sets and offer sets; we denote this operator by a comma placed in infix position: $\mathcal{S}, \mathcal{S}' \triangleq \langle \mathcal{R} \cup \mathcal{R}' : \mathcal{O} \cup \mathcal{O}' \rangle$, if $\mathcal{S} = \langle \mathcal{R} : \mathcal{O} \rangle$ and $\mathcal{S}' = \langle \mathcal{R}' : \mathcal{O}' \rangle$. Needless to say, ready set union is commutative and associative: $\mathcal{S}, \mathcal{S}' = \mathcal{S}', \mathcal{S}$ and $(\mathcal{S}, \mathcal{S}'), \mathcal{S}'' = \mathcal{S}, (\mathcal{S}', \mathcal{S}'')$. We also use a unary operator $\mathcal{S}^{\backslash a}$, that suppresses all offers made on the name a: if $\mathcal{S} = \langle \mathcal{R} : \mathcal{O} \rangle$, then $\mathcal{S}^{\backslash a} \triangleq \langle \mathcal{R} : \mathcal{O} \setminus \{a^c, a^p, \overline{a}^c, \overline{a}^p\} \rangle$. We define another unary operator on prefix sets, that complements any action in the set: $\overline{\mathcal{R}} = \{\overline{a}^r \mid \exists r \in \{c,p\}.\ a^r \in \mathcal{R}\} \cup \{a^r \mid \exists r \in \{c,p\}.\ \overline{a}^r \in \mathcal{R}\}$. We note the empty ready set by $\langle\rangle \triangleq \langle \emptyset : \emptyset \rangle$. We finally define a partial order on ready sets, noted $\mathcal{S}' \preceq \mathcal{S}$, that is equivalent to the proposition $\mathcal{O}' \subseteq \mathcal{O} \wedge \mathcal{R} \subseteq \mathcal{R}'$. Ready sets are hence ordered contravariantly in their sets of requests and offers: when $\mathcal{S}' \preceq \mathcal{S}$, $\mathcal{S}'$ has less offers but more requests than $\mathcal{S}$. Similar ready sets, though devised in different purposes, can be found in various contributions in respectively untimed [BIM95] and timed settings [JLS00, ABL98].

We define our process semantics in two steps, in the style originated by Berry and Boudol with their *chemical abstract machine* [BB92], and later adopted by Milner (see for example [Mil92] or [Mil93]). That is, we first give a *structural congruence* relation over processes, reduction rules then being given *modulo* term reorganization using the congruence (by implicit application of the rule CONV in Table 3.2). Our congruence relation does not depart much from the existing congruences for the π-calculus; it is defined in two steps, starting with a "ground" version $\equiv^g$ defined as the smallest equivalence relation closed under the rules of Table 3.1. One of the primary goals of this congruence is to provide a proper treatment of the idle process: placed in any context, the idle process may never impeach the progress of the whole system. Hence, 0 is the neutral element for choice and composition, while restriction and matching in front of 0 can be safely eliminated. Having those rules otherwise allows us to avoid letting 0 perform any transition (even time-passing ones), that would complicate the theory in a rather irrelevant fashion, since all 0 processes are indistinguishable anyway. Hence, there is no operational rule treating 0 in Tables 3.2 and 3.3.

In Table 3.1 are then given the classical rules stating that the choice and composition operators are commutative and associative. The matching operator may not stop the process as in the π-calculus, but instead behaves as 0 when the test is false. The last rule is the usual rule on α-conversion of terms. Note that this rule

only applies to port names: α-conversion is not powerful enough to cater for clock names. A clock name indeed becomes free when the corresponding clock is set to 0, and α-conversion may not be applied hereafter. This is embarrassing, because two processes that differ *only* on the name of *one* clock but are identical in every other way (including the value associated to the clock name on which they disagree) behave in the same way.

$$(t+0)\rho \equiv^g t\rho \quad (t \mid 0)\rho \equiv^g t\rho \quad (\nu n\ 0)\rho \equiv^g 0\,\rho \quad ([a=b]0)\rho \equiv^g 0\,\rho$$
$$(t+u)\rho \equiv^g (u+t)\rho \quad ((t+u)+v)\rho \equiv^g (t+(u+v))\rho$$
$$(t \mid u)\rho \equiv^g (u \mid t)\rho \quad ((t \mid u) \mid v)\rho \equiv^g (t \mid (u \mid v))\rho$$
$$([a=a]t)\rho \equiv^g t\rho$$
$$([a=b]t)\rho \equiv^g 0\,\rho \textbf{ if } b \textbf{ is syntactically different from } a$$
$$(\nu a\ t)\rho \equiv^g (\nu b\ u)\rho \textbf{ if } t[c/a] = u[c/b] \textbf{ for some name } c \textbf{ fresh in } t \textbf{ and } u$$

Table 3.1: The Structural Congruence for π^δ

The definition of the clock-resetting operation $\rho^{\downarrow x}$ in Section 3.4 required that $x \notin dom(\rho)$. This has for consequence that a new name is introduced in ρ each time a clock x is set to zero. Over processes with recursion-free terms, this could be a cause of incompletion for our proof system. Over recursion-enabled terms (for example reactive processes, which execution never ends), this would be the source of an obviously artificial ever-going expansion of the state space during execution. Furthermore, it may be easily noticed that, unlike port names which are not meaningful when bounded (acting only as placeholders) but meaningful when free (they then stand for uniquely defined constants), clock names are *never* meaningful! A free clock name can indeed be seen as a variable, standing for the value associated to its name by the process context. Hence the incapacity of α-conversion to solve this problem. Needing a more powerful structural congruence $\equiv$ over processes, we define it by extending the ground relation $\equiv^g$ above with an additional syntactic way of identifying processes. Informally, if there exists an isomorphism of free clock names between the clock contexts of two processes, then those processes are deemed equivalent. Formally, for two terms t, u and two valuations ρ, κ with $\mathrm{fc}(t) \subseteq dom(\rho)$ and $\mathrm{fc}(u) \subseteq dom(\kappa)$, if we take $\mathrm{fc}(t)$ and $\mathrm{fc}(u)$ to be arbitrarily ordered respectively as $x_1, x_2, \cdots, x_k$ and $y_1, y_2, \cdots, y_k$, then

$$t\,\rho \equiv u\,\kappa \Leftrightarrow \left\{ \begin{array}{l} t\,\rho \equiv^g u\,\kappa \ \wedge \\ \exists z_1, z_2, \cdots, z_k. \end{array} \right. \left\{ \begin{array}{l} set(\tilde{z}) \cap (set(\tilde{x}) \cup set(\tilde{y})) = \emptyset \ \wedge \\ \rho^{/\mathrm{fc}(t)}[\tilde{z}/\tilde{x}] = \kappa^{/\mathrm{fc}(u)}[\tilde{z}/\tilde{y}] \end{array} \right.$$

with for all $i, j \in \{1..k\}$, $i \neq j$, z_i, z_j pairwise different. We bluntly name this way of converting processes *free clock conversion*. Requiring the clocks to be arbitrarily ordered is harmless: it amounts to finding a one-to-one correspondence between names *without taking into account their syntactical value*. This is a household way of reasoning about and implementing formal logics; De Bruijn indices [dB72] are here the conspicuous syntactic sieve to be used, leading to the intended result in a

straightforward and very efficient fashion. Those indices are for instance applied to π-calculus terms in the *mobility workbench* [VM94].

We now proceed to the examination of the operational rules themselves, shown in Tables 3.2 and 3.3. The deduction rules are two-part schemata which informal interpretation can be that for some process $t\rho$ the reduction $t\rho \xrightarrow{\lambda} t'\rho'$ at the bottom part (the consequent) can be performed if the reduction(s) in the top part (the antecedent) can be performed. The process $t\rho$ is then given a TTS in which transition λ has for origin the root of the transition system, and for destination the root of the TTS associated to the process $t'\rho'$. Axioms PRE, TNSEL, TSEL, TURG and TMISS do not have one or many reductions as antecedent, but a statically decidable condition on the clock values in context ρ. The rules are divided in two parts: the rules very-similar to the "traditional" ones found in π-calculus theories are grouped in Table 3.2, while time-related rules are grouped in Table 3.3. We give the rules yielding the so-called *late* semantics for name instantiation; for more insights on late and early semantics, see for example [MPW93].

We give the informal meaning of each rule, starting by the ones of Table 3.2. The axiom PRE states that a process with a prefixed control part may perform the prefix action π if its context ρ satisfies the selection condition σ. The set of reachable clock valuations $reach(\rho, \pi)$ from valuation ρ through transition π is defined by: $reach(\rho, \pi) \triangleq \{\kappa \mid dom(\kappa) = (dom(\rho) \cup \{x\}) \wedge \kappa^{\setminus x} = \rho\}$ if $\pi = a(x)$ for some a, and $reach(\rho, \pi) \triangleq \{\rho\}$ otherwise. Hence, the process context can be extended in an infinite (non recursively-enumerable) number of ways should the value of a clock be received. This is logical, since the process has no way to know in advance which value will be provided by its environment.

Rule COM allows the *sort-respecting* communication of a free name n over port a, producing a silent transition. Free occurrences of m in t are then replaced by n, effectively simulating communication of n, that becomes free in both u and t. This rule generalizes the one found in previous π-calculus theories by allowing the sending of free clock names. This explains the side condition on the rule, that copes with context management: if there was already a free clock named n in t, its value must be replaced by the newly received one. Hence we require that the valuation $\rho'^{\setminus n}\kappa'$ be defined.

Associated, the rules OPEN and CLOSE can be used to transmit the bound name b over the port a. To denote this fact, the OPEN rule employs the *bound output* prefix $\overline{a}(b)$ that has been introduced in the definition of λ. Proof-theoretically, the rules PRE and OPEN can be used to provide the antecedent to rule CLOSE. After this latter rule has been applied, the name b is hidden in the resulting process. The side condition requires that processes which control parts are put in parallel must have compatible contexts.

The SUM and PAR rules are the classical rules for choice and parallel composition. They can be applied only to process interaction, the corresponding time-passing rules TSUM and TPAR are shown in Table 3.3. The rule RES states that a name declaration does not influence a transition that does not involve the declared name.

The last three rules are special because they allow to infer transitions with label λ, ranging over delays as well as actions. The rule RESET, read bottom-up, states that one may deduce that $(\nu x\ t)\rho \xrightarrow{\lambda} t'\rho'$ when the same control part with environment

$$\text{PRE}\ \frac{\rho \vDash \sigma \quad \rho' \in reach(\rho,\pi)}{([\sigma,\upsilon]\pi.\,t)\rho \xrightarrow{\pi} t\rho'} \qquad \text{COM}\ \frac{t\rho \xrightarrow{a(m)} t'\rho' \quad u\kappa \xrightarrow{\overline{a}\,n} u'\kappa'}{(t|u)\rho \xrightarrow{\tau} (t'[n/m] \mid u)\rho'^{\backslash m}\kappa'}\,\nabla$$

$$\text{OPEN}\ \frac{t\rho \xrightarrow{\overline{a}\,b} t'\rho'}{(\nu b\ t)\rho \xrightarrow{\overline{a}(b)} t'\rho'}\ b \neq a \qquad \text{CLOSE}\ \frac{t\rho \xrightarrow{a(b)} t'\rho' \quad u\kappa \xrightarrow{\overline{a}(b)} u'\kappa'}{(t \mid u)\rho\kappa \xrightarrow{\tau} ((b)(t' \mid u'))\rho'\kappa'}\,\Delta$$

$$\text{SUM}\ \frac{t\rho \xrightarrow{\pi} t'\rho'}{(t+u)\rho \xrightarrow{\pi} t'\rho'}\,\Diamond \qquad \text{PAR}\ \frac{t\rho \xrightarrow{\pi} t'\rho'}{(t \mid u)\rho \xrightarrow{\pi} (t' \mid u)\rho'}\,\blacklozenge$$

$$\text{RES}\ \frac{t\rho \xrightarrow{\pi} t'\rho'}{(\nu a\ t)\rho \xrightarrow{\pi} (\nu a\ t')\rho'}\ a \notin \mathrm{n}(\pi) \qquad \text{RESET}\ \frac{t\rho^{\downarrow x} \xrightarrow{\lambda} t'\rho'}{(\nu x\ t)\rho \xrightarrow{\lambda} t'\rho'}\ x \notin dom(\rho)$$

$$\text{ID}\ \frac{t\,[\tilde{n}/\tilde{m}]\,\rho \xrightarrow{\lambda} t'\rho'}{(A(\tilde{n}))\rho \xrightarrow{\lambda} t'\rho'}\ A(\tilde{m}) \triangleq t \qquad \text{CONV}\ \frac{u\kappa \equiv t\rho \quad t\rho \xrightarrow{\lambda} t'\rho' \quad u'\kappa' \equiv t'\rho'}{u\kappa \xrightarrow{\lambda} u'\kappa'}$$

$\Delta \triangleq \rho\kappa$ and $\rho'\kappa'$ defined. $\Diamond = dom(\rho) \supseteq \mathrm{fc}(t) \cup \mathrm{fc}(u)$.
$\nabla \triangleq \rho\kappa$ and $\rho'^{\backslash m}\kappa'$ defined, $(m \in \mathbf{P}) \Leftrightarrow (n \in \mathbf{P})$. $\blacklozenge \triangleq \Diamond \wedge ((\mathrm{bn}(\pi) \cap \mathrm{fn}(u)) = \emptyset)$.

Table 3.2: Late Transitional Semantics for π^δ: part 1

$$\text{TNSEL}\ \frac{\forall \delta' \in (0,\delta)\ \rho^{+\delta'} \vDash \neg\,\sigma}{([\sigma,\upsilon]\pi.\,t)\rho \xrightarrow{\delta\langle\rangle} ([\sigma,\upsilon]\pi.\,t)\rho^{+\delta}} \qquad \text{TSUM}\ \frac{t\rho \xrightarrow{\delta\mathcal{S}} t'\rho^{+\delta} \quad u\kappa \xrightarrow{\delta\mathcal{S}'} u'\kappa^{+\delta}}{(t+u)\rho\kappa \xrightarrow{\delta\mathcal{S},\mathcal{S}'} (t'+u')(\rho\kappa)^{+\delta}}\,\Delta$$

$$\text{TSEL}\ \frac{\forall \delta' \in [\,0,\delta)\ \rho^{+\delta'} \vDash \sigma \wedge \neg\,\upsilon}{([\sigma,\upsilon]\pi.\,t)\rho \xrightarrow{\delta\langle\emptyset:s(\pi)\rangle} ([\sigma,\upsilon]\pi.\,t)\rho^{+\delta}} \qquad \text{TPAR}\ \frac{t\rho \xrightarrow{\delta\mathcal{S}} t'\rho^{+\delta} \quad u\kappa \xrightarrow{\delta\mathcal{S}'} u'\kappa^{+\delta}}{(t \mid u)\rho\kappa \xrightarrow{\delta\mathcal{S},\mathcal{S}'} (t' \mid u')(\rho\kappa)^{+\delta}}\,\Box$$

$$\text{TURG}\ \frac{\forall \delta' \in [\,0,\delta]\ \rho^{+\delta'} \vDash \upsilon}{([\sigma,\upsilon]\pi.\,t)\rho \xrightarrow{\delta\langle s(\pi):\emptyset\rangle} ([\sigma,\upsilon]\pi.\,t)\rho^{+\delta}} \qquad \text{TRES}\ \frac{t\rho \xrightarrow{\delta\mathcal{S}} t\rho^{+\delta}}{(\nu a\ t)\rho \xrightarrow{\delta\mathcal{S}\backslash a} (\nu a\ t)\rho^{+\delta}}$$

$$\text{TMISS}\ \frac{ToError(\upsilon)}{([\sigma,\upsilon]\pi^\star.\,t)\rho \xrightarrow{\delta\langle s(\pi^\star)\rangle} Error\,\rho^{+\delta}} \qquad \text{TERR}\ \frac{ProgErr(\upsilon)}{Error^\upsilon\rho \xrightarrow{\delta\langle\rangle} Error^\upsilon\,\rho^{+\delta}}$$

$\Delta \triangleq \rho\kappa$ defined. $s(\pi) \triangleq \mathrm{subject}(\pi)^{sort(object(\pi))}$.
$ToError(\upsilon) \triangleq (\rho^{+\delta} \vDash \neg\,\upsilon) \wedge (\forall \delta' \in [\,0,\delta)\ \rho^{+\delta'} \vDash \upsilon)$.
$ProgErr(\upsilon) \triangleq (\forall \delta' \in [0,\delta]\ \rho^{+\delta'} \vDash \neg\upsilon) \vee (ToError(\neg\upsilon))$.
$\Box \triangleq \Delta \wedge (\mathcal{S} = \langle \mathcal{R} : \mathcal{O}\rangle \wedge \mathcal{S}' = \langle \mathcal{R}' : \mathcal{O}'\rangle) \Rightarrow (\overline{\mathcal{R}} \cap \mathcal{O}' = \emptyset \wedge \overline{\mathcal{R}'} \cap \mathcal{O} = \emptyset \wedge \overline{\mathcal{R}} \cap \mathcal{R}' = \emptyset)$

Table 3.3: Late Transitional Semantics for π^δ: part 2

$\rho^{\downarrow x}$ (where x is set to 0) can also perform the action, leading to $t'\rho'$. The valuation ρ' is obtained from $\rho^{\downarrow x}$ by letting the transition happen: it *has* to take account of the clock reset as wanted. The side condition on the rule ensures that the process has been properly free-clock-converted so that x is not already in the domain of ρ. Process instantiation behaves as the term associated to the name A with the arguments $\tilde{n}$ replacing the formal parameters $\tilde{m}$, in the same context ρ. The rule CONV for term reorganization through structural congruence is as usual.

All the rules in Table 3.3 aim at letting a delay δ elapse; as mentioned previously we do not allow transitions to pass a null delay, hence we assume that $\delta > 0$. We first review the axioms: TNSEL allows a prefixed process to let time pass freely as long as the selection condition for π is false. The interval $(0, \delta)$ for δ' is left-open in the axiom antecedent because σ can be initially true, if π was enabled and becomes forbidden. The axiom TSEL settles the case where a prefix action is selected but not urgent; the right-open interval for δ' allows time to pass until either the action becomes urgent or it is deselected.

When the action is urgent, then both axioms TURG and TMISS may apply. The axiom TURG yields time-passing transitions that keep the urgency condition true (the interval for δ' is left-right closed). Oppositely, TMISS tries to lead the whole system to an error state. To avoid this, the only solution for the environment is to impeach time progression by either proposing a matching offer to this requirement or another (many processes in parallel can exhibit urgency conditions at the same time) or by having itself an urgency condition on some action which has an immediate match. In any other case, the process $Error$ is reached. Of course, the silent input prefixes, of the form $[\sigma, \upsilon]\tau.\, t$, are not concerned by this rule since τ actions are not a way of achieving synchronization among processes. Hence, the prefixes allowed by rule TMISS range over $\pi^\star$, which does not contain τ. The rule TERR allows time progression until υ becomes true, if this is not already the case. Exactly when υ becomes true, the process evolves to $Error$. If υ never becomes true, time may progress unboundedly.

The rules TSUM and TPAR show how offers and requirements assemble when involved in a choice or when composed in parallel. The essential point is that the two operators actually behave in the same way regarding time, except that the side condition on the parallel rule forbids the conjoining of ready sets that carry complementary actions. Finally, applying the rule TRES forbids communication with the environment on the port a. Hence, it keeps the requirements (although those requirements, being hidden, cannot be satisfied by the environment) and suppresses all offers on a.

3.7 Semantics of A Small Example

To illustrate how the rules can be worked-in, we give a small example of the possible transitions for two very simple communicating processes. We start by settling some common notational conveniences that we shall use from now on. First, we shall generally omit trailing 0's in processes, writing $[\sigma, \upsilon]\pi$ instead of $[\sigma, \upsilon]\pi.\, 0$. Second, we allow the use of $\pi.\, t$ for $[\top, \bot]\pi.\, t$. This has the nice feature announced earlier

that untimed π-calculus terms are straightforwardly meaningful in our calculus. Third, we sometimes gather lists of consecutive declarations into one containing many names: $(\nu m, n, o)\ t \triangleq \nu m\ \nu n\ \nu o\ t$.

The term we are about to examine is the following:

$$t \equiv \nu a\ (\nu b\ \overline{a}b.\, \nu y\ [y \leq 6, y < 6]b() \mid a(c).\, \nu x\ [x \geq 5, \bot]\overline{c}())\ .$$

To form a process P, we consider t adjoined by any context ρ. We divide the term t in two subterms $\nu b\ \overline{a}b.\, u$ and $a(c).\, v$ with $u \equiv \nu y\ [y \leq 6, y < 6]b()$ and $v \equiv \nu x\ [x \geq 5, \bot]\overline{c}()$. Those terms are put in parallel and communicate using a private port a. As t contains no free name occurrence, and since clocks x and y are properly set to 0 before being used, the behavior of P is independent from the initial clock context ρ. The sub-process $\nu b\ \overline{a}b.\, u$ creates a new port b and waits indefinitely for its environment to be ready to receive b over port a. If the synchronization happens, the continuation u immediately sets clock y to 0 and begins waiting for input on b. As the input on b is urgent, u is eager to synchronize, and if no output on b has been proposed after exactly 6 time units, an error is produced.

On the other side, the process $a(c).\, v$ expects an input on port a, naming the to-be-received argument c. It then sets clock x to 0, and may let five time units pass without doing anything, after what it agrees to output on port c for an indefinite amount of time.

Now, formally, here follow a few examples of deductions allowing some transitions. Initially, P can let any amount of time pass, meaning that whatever δ, there exists ρ_u and ρ_v so that $\rho = \rho_u \rho_v$ is defined and we can write:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall \delta' \in [0,\delta)\ \rho_u^{+\delta'} \vDash \top\ \wedge \neg\bot}{(\overline{a}b.\, u)\rho_u \xrightarrow{\delta\langle\emptyset:\overline{a}^p\rangle} (\overline{a}b.\, u)\rho_u^{+\delta}}\ \text{TSEL}}{(\nu b\ \overline{a}b.\, u)\rho_u \xrightarrow{\delta\langle\emptyset:\overline{a}^p\rangle} (\nu b\ \overline{a}b.\, u)\rho_u^{+\delta}}\ \text{TRES} \qquad \dfrac{\dfrac{\forall \delta' \in [0,\delta)\ \rho_v^{+\delta'} \vDash \top\ \wedge \neg\bot}{(a(c)\ v)\rho_v \xrightarrow{\delta\langle\emptyset:a^p\rangle} (a(c).\, v)\rho_v^{+\delta}}\ \text{TSEL}}{(\nu b\ \overline{a}b.\, u)\rho_u \mid (a(c).\, v)\rho_v \xrightarrow{\delta\langle\emptyset:a^p,\overline{a}^p\rangle} (\nu b\ \overline{a}b.\, u \mid a(c).\, v)\rho^{+\delta}}\ \text{TPAR}}{t\rho \xrightarrow{\delta\langle\rangle} t\rho^{+\delta}}\ \text{TRES}$$

The above reasoning tells that, in the case where an interaction is not required by any of the two parties, its occurrence depends only on the good will of what we can call the "universe" or, more pragmatically, the *execution machine*. We can now show that an interaction on a is possible after any number of time-passing transitions. The proof is rather identical to what it would be in the π-calculus: we can deduce that u may reduce performing a bound output of c on a (which it does not actually contain because in u the name is b) by using the CONV rule to α-convert c to b. The the counterpart on the v side is trivial; if we suppose $u' \equiv \nu y\ [y \leq 6, y < 6]c()$, we can write:

$$\text{RES}\ \dfrac{\text{CLOSE}\ \dfrac{\text{OPEN}\ \dfrac{\text{PRE}\ \dfrac{\rho_u \vDash \top}{(\overline{a}c.\, u')\rho_u \xrightarrow{\overline{a}c} u'\rho_u}}{(\nu c\ \overline{a}c.\, u')\rho_u \xrightarrow{\overline{a}(c)} u'\rho_u} \quad \text{CONV}}{(\nu b\ \overline{a}b.\, u)\rho_u \xrightarrow{\overline{a}(c)} u'\rho_u} \qquad \dfrac{\text{PRE}\ \dfrac{\rho_v \vDash \top}{(a(c).\, v)\rho_v \xrightarrow{a(c)} v\rho_v}}{(\nu b\ \overline{a}b.\, u \mid a(c).\, v)\rho \xrightarrow{\tau} (\nu c\ (u' \mid v))\rho}}{t\rho \xrightarrow{\tau} \nu a\ (\nu c\ (u' \mid v))\rho}$$

We now address the more interesting case where, after the initial interaction on a, both process terms u' and v order a clock reset and then allow time-passing transitions that exhibit offers and requirements. One of those transitions, with $\delta < 5$, can be deduced by:

$$\text{TRES}\ \dfrac{\text{TPAR}\ \dfrac{u'\rho_u \xrightarrow{\delta\langle c:\emptyset\rangle} u'\rho_u^{\downarrow y+\delta} \quad v\rho_v \xrightarrow{\delta\langle\rangle} v\rho_v^{\downarrow x+\delta}}{(u' \mid v)\rho \xrightarrow{\delta\langle c:\emptyset\rangle} (u' \mid v)\rho^{\downarrow x\downarrow y+\delta}}}{(\nu a\ (u' \mid v))\rho \xrightarrow{\delta\langle\rangle} (\nu a\ (u' \mid v))\rho^{\downarrow x\downarrow y+\delta}}$$

because

$$\text{RESET}\ \dfrac{\text{TURG}\ \dfrac{\forall \delta' \in [0,\delta]\ \rho_u^{\downarrow y+\delta'} \vDash y < 6}{([y \le 6, y < 6]c())\rho_u^{\downarrow y} \xrightarrow{\delta\langle c:\emptyset\rangle} ([y \le 6, y < 6]c())\rho_u^{\downarrow y+\delta}}}{u'\rho_u \xrightarrow{\delta\langle c:\emptyset\rangle} u'\rho_u^{\downarrow y+\delta}}$$

and

$$\text{RESET}\ \dfrac{\text{TNSEL}\ \dfrac{\forall \delta' \in (0,\delta)\ \rho_v^{\downarrow x+\delta'} \nvDash x \le 5}{([x \ge 5, \bot]\overline{c}())\rho_v^{\downarrow x} \xrightarrow{\delta\langle\rangle} ([x \ge 5, \bot]\overline{c}())\rho_v^{\downarrow x+\delta}}}{v\rho_v \xrightarrow{\delta\langle\rangle} v\rho_v^{\downarrow x+\delta}}$$

After several time progressions $\delta_1, \delta_2, \cdots, \delta_k$ such that the sum $\delta_1 + \delta_2 + \cdots + \delta_k = 5$, then time cannot progress anymore because of the urgency condition in u'. Indeed, the side condition in rule TPAR, that was true in the previous proof tree, is not true anymore since u' requires interaction on c as before, and v offers interaction on c starting when x equals 5. The only available transition is therefore the one where u' and v communicate.

The final question is: can this process lead to an error? From the previous reasoning, one should by intuition answer "no". But we can give a perfect proof-theoretic answer: suppose we try to make the process trigger a transition leading to $y = 6$ for example. Then, the rule TMISS can be invoked for u'. However, though v is ready to accept to let such a delay elapse, the transition cannot be inferred from the rules of Table 3.3, again because of the side condition on rule TPAR.

3.8 Model Properties Over Time

We now discuss some important properties of the transition systems generated by our processes. This discussion includes a comparison with the properties exhibited by other timed process algebras and the π-calculus.

First, let us remark that our processes are *infinite-state* in many regards. By combining recursion with restriction and composition, one may indeed create terms that have an infinite number of parallel processes as well as an infinite number of names. This is true in all π-calculus theories, since the π-calculus is actually Turing-expressive: the λ-calculus can easily be encoded in it [Mil92]. Having clocks which values range over a dense time domain yields however a semantically conspicuous property: for any term t, there are an infinite (even non recursively-enumerable) number of processes $t\rho$.

This has the also unusual consequence for a π-calculus theory that our transition systems are *infinitely branching*: from a given process, an infinite number of time-passing transitions are possible. Our processes not even retain an essential property of other π-calculus theories, named *image-finiteness*: for a given term and a given transition label, a potentially *infinite* number of processes are reachable. The PRE rule can easily be judged guilty for that, a process that inputs a clock value reaching an infinite number of processes through this transition. If we preclude clock value transmission however, the property is restored and can be proved as usual on the ground that, though the argument to an input transition ranges over an infinite number of (port) names, there can be only finitely many names that can be effectively proposed; see for example [SW01, page 45] for details.

From the rules and axioms of Tables 3.2 and 3.3, one can deduce a several other properties. *Time determinism* is common to all the timed process algebras that we now of. It is retained by models of our theory, for which it can be written:

$$\forall P, P', P'', \delta, \mathcal{S}.\ P \xrightarrow{\delta\langle\mathcal{S}\rangle} P' \wedge P \xrightarrow{\delta\langle\mathcal{S}\rangle} P'' \Rightarrow P' = P'' \ .$$

We can also easily prove that, between any two points reachable with the same ready set, our transition systems are *interval-trajectoried*:

$$\begin{aligned}\forall P, P_b, P_e, \delta_b, \delta_e, \mathcal{S}.\ \ & (\delta_b < \delta_e \wedge P_b, P_e \neq \mathit{Error} \wedge P \xrightarrow{\delta_b\langle\mathcal{S}\rangle} P_b \wedge P \xrightarrow{\delta_e\langle\mathcal{S}\rangle} P_e) \Rightarrow \\ & (\forall \delta, \delta' \in [\delta_b, \delta_e].\ (\delta < \delta') \Rightarrow (\exists P', P''.\ P \xrightarrow{\delta\langle\mathcal{S}\rangle} P' \xrightarrow{(\delta'-\delta)\langle\mathcal{S}\rangle} P''))\end{aligned}$$

This means that each point of the interval $[\delta_b, \delta_e]$ is reachable, and that from it any later point in the same interval is reachable too. However, it is false that in our transition systems all time progressions are trajectoried; a good reason for that is that we *do not* have the *time continuity* property [NS92, Yi91], that is usually defined by:

$$\forall P, P', P'', \delta, \delta'.\ P \xrightarrow{\delta} P' \wedge P' \xrightarrow{\delta'} P'' \Rightarrow P' \xrightarrow{\delta+\delta'} P'' \ .$$

Indeed, since we are using ready sets, time continuity should be formulated as:

$$\forall P, P', P'', \delta, \mathcal{S}, \mathcal{S}'.\ P \xrightarrow{\delta\langle\mathcal{S}\rangle} P' \wedge P' \xrightarrow{\delta\langle\mathcal{S}'\rangle} P'' \Rightarrow \exists \mathcal{S}''.P \xrightarrow{(\delta+\delta')\langle\mathcal{S}''\rangle} P''$$

which is obviously false. Since having trajectoried time-passing transitions implies the time continuity property for those transitions [JSV93], we deduce that our transitions are not trajectoried either.

Further considerations could be done over some other time properties of our models; they *do not* have neither the weak *interval persistency* of [NSY93], nor the stronger *unlimited persistency* of [Yi91], defined by:

$$\forall P, Q, P', \delta, \pi.\ P \xrightarrow{\pi} P' \wedge P \xrightarrow{\delta} Q \Rightarrow \exists P''.\ Q \xrightarrow{\pi} P'' \ .$$

The last property usually considered in other timed process algebras is the *maximal progress* property. It states that certain actions (often the *silent* action) may occur *as soon as possible*. Our formal treatment and the operators we introduce in our language allow more distinction regarding urgency properties than other process algebras. Maximal progress is *not* a property of our models, but we shall give a proper justification for this when comparing our calculus to others, in Section 3.9.

As almost all other models obtained with timed process algebras, our models can exhibit *deadlocks* and *livelocks*, and produce *zeno executions* [R. 91, BGS00, Tri99]. Occurrence of deadlocks in distributed programs and their specifications is one of the earlier problems identified in computer science (see for example [Dij76, SM73] for a later account). Its definition is that the system under study cannot accomplish any further transitions, staying locked in its progress. In our model, the deadlocked process is $Error$; it can be reached for instance when an unsatisfied requirement is encountered. Livelocks are less pathological behaviors: time may continue to progress, but no other action than time-passing ones will ever happen. This is for example the case when many processes put in parallel do not have input or output transitions that may become available in the future. Finally, after the Greek philosopher Zeno of Elea (490-425 B.C.) that produced the well-known paradox about Achilles and the turtle, the adjective "zeno" designates infinite executions where time does not diverge: the sum of delays elapsed along the execution tends towards a finite upper-bound. A possible source of zenoness is the occurrence of an infinite number of actions in a finite time during an execution. In general, zeno executions should be absolutely avoided because they represent behaviors impossible in real life.

We will propose solution to avoid deadlocking and zeno executions in the Chapter 5. However, livelocks are not judged really pathological in our theory, and can not be directly prevented. Indeed, if a process proposes only non-urgent interaction to its environment, then this environment has the right to refuse them. The process may then let time pass up to the moment where none of the prefixes of its current terms are selectable anymore. There are however means to avoid such situations, one of them being to use urgent transitions labeled with the silent action.

3.9 Discussion and Related Works

We essentially have two points to discuss: the expressiveness of the operators used in our language, and the properties of the resulting models (*i.e.* terms).

A good comparison of several timed process algebras cited in this thesis can be found in [NS92]; this comparison covers both the operators used in those algebras and the models associated to their terms. Our language is more powerful than all the timed process algebras that we know of because it allows:

- the transmission of port names to simulate mobility,
- the transmission of clock values,
- the use of urgency conditions on prefixes which subject name is *free*.

We do not elaborate more on the first two points because they reflect the existence of the well-known "gap" of expressiveness between CCS and the π-calculus; previously existing timed process algebras being based on either CCS, ACP or CSP, this expressiveness gap logically appears in the timed case. The third point deserves more attention because it has strong philosophical consequences on the meaning of programs and their comparison (the associated notion of equivalence studied in Chapter 4), as well as their composition and their verification (both studied in Chapter 5).

First, we need to give more technical details on process control and composition: delving into such details would have been inappropriate in the introduction of this chapter, but it is now necessary, if not fully relevant.

The main issue is whether one should write only *input enabled* and *machine realizable* specifications or not. A process is input enabled if the set of messages it is able to receive remains constant, this set forming a *uniform service interface*: any of the messages it contains can be received at any moment. When processes specify only *safety* properties [AS85], having uniform service interfaces is enough to guarantee the autonomy of processes, each of them being in complete control of its output actions. If however processes are able to specify *liveness* properties [AS85], then input enabling has to be lifted-up to *machine realizability* to confer processes the proper control of their actions.

A specification is (machine) realizable [ALW89] if and only if there exists an implementation of it (*i.e.* a specification concrete enough so as to be called a *program*) for which any finite execution can be extended to an infinite execution. In other words, machine realizability imposes that no safety property may prevent the realization of any liveness property. For process terms with no free variables (*i.e.* that form *closed* systems), this property is called *machine closure*. For *open* systems, machine realizability is called *receptiveness* [Dil89b]: at any point in time, the process must be able to progress whatever its environment does.

As stated in the introduction to this chapter, "traditional" untimed process algebras feature processes exhibiting non-uniform service interfaces. Those processes specify only safety properties; an extension of CCS with fairness assumptions has been introduced in [Par85] but the autor did not address control and composition-related issues. Allowing only input enabled and machine realizable specifications has been mainly advocated in [ALW89] and [LT87]. Those contributions adopt the view that a process should never constrain its environment by refusing some input, so that processes have complete control over their own output actions. This is justified by the fact that a program running in some real-life system has indeed no control over

the actions performed by its environment (for example the computer executing the program). Among several benefits, machine realizability provides a compositional semantics for fair automata that allows for top-down modular refinement in [LT87].

In the case of timed specifications, action control and machine realizability are even more sensitive issues. Indeed, if the most general kind of urgency condition is allowed (as in the language presented in this chapter), then a process can force the system to deadlock under some condition requiring that its enviromnent be ready to interact on a given port before a given deadline; such a property is typically a *bounded liveness* property. Such process can not be machine realizable. Are not realizable also the processes that exhibit zeno executions, because by preventing time to diverge they forbid their environment to preform later due actions.

Some timed process algebras, such as TCCS [Yi91], TPL [HR95], Timed CSP [Sch95], RT-Lotos [CdO95] and ET-Lotos [LL92], try to limitate the contraining power of processes over their environment. More specifically, they allow only silent or hidden actions to be urgent. They do not however completely address the issue, since they consider only finite (though unbounded in length) executions, and forget about machine realizability. Preventing urgency conditions to occur in certain situations is indeed not sufficient to guarantee the realizablility of a specification: the actual criterion is more demanding [TMM88, GSSAL94, AH97].

Other timed process algebras explicitly allow processes to impose urgency conditions on actions that are visible by their environment: we can cite at least TeCCS [Mol90], ATP [NSY93], real-time ACP [BB91] and urgent-LOTOS [BL92a, BL92b]. The corresponding specifications are therfore unrealizable, but this point is not justified, even not addressed at all, in the referred articles. In the first three of those process algebras, urgency conditions come under the form of *time-lock* prefixes: some prefixes, that we write for example $a.t$, do not allow time to progress. The only way for the environment to unlock the system is then to interact with $a.t$ on a.

The urgent LOTOS of Bolognesi and Lucidi makes use of a more powerful and subtle urgency operator. In urgent LOTOS, interactions on a port `a` are not urgent, unless an ugency operator $\texttt{asap}_{\texttt{a}}$ is placed on top of the interacting processes. An urgency condition is then only respected by processes *under* it: if `a` is free in $\texttt{asap}_{\texttt{a}}\, t$ and can therefore be used as a synchronization port between this process and its environment, the urgency condition can not "see" whether the environment proposes a at some point or not, and it hence can not make the urgency condition respected by the environment.

Our operator for urgency can be seen an extension of the `asap` operator of [BL92a] that guarantees the respect of an urgency condition even by the environment of the given process. Our operator is thus more powerful than all the other urgency operators we have seen in [Mol90, NSY93, BB91, BL92a]. Regarding realizability issues this seems of little importance since all those operators are *too powerful* anyway! Furthermore, having this more prowerful operator seems also not to be a bigger problem regarding composition-related issues: it is hopeless to obtain a compositional model at no cost, as done in the I/O automata model, when realizability is relinquished.

But then, what is the interest of reasoning on non-realizable specifications? How can we perform compositional reasoning on those specifications, since this was an-

nounced as a primary goal in the introduction to this chapter?

We answer to the first question by another question. Why indeed should only realizable processes be worth of insterest? The advocated approach in the literature on compositional verification (*e.g.* [LT87, AL93, AL94, AL95, TMM88, GSSAL94, AH99, AH97]) is to build realizable systems by composing smaller realizable components. Hence realizability is preserved by composition. We claim that interesting results can be devised on non-realizable open processes, processes that can be used afterwards in realizable systems. If indeed a process is not receptive, we can still study its properties independently; this study will of course be useful only if there exists an enviroment for this process that makes the composition (process + environment) machine realizable (or machine closed).

In other words, we allow the composition of non-realizable specifications, and we aim at recovering properties obtained when composing only realizable specifications. Of course, both methods have a certain cost. To compose realizable specifications, one has to previously check that the to-be-composed specifications are indeed realizable. In I/O automata, this check is simplified by the structure of the automata, and it boils down to checking that the composed automata have disjoint output sets. We will propose in Chapter 5 a type-based static analysis method that imposes components put in parallel to have *compatible* behaviors. Compatibility checking assures that one process requiring to perform an inputs or an output action on some port will be composed only with a good-willing environment. We think that this method is more tractable than realizability checking, since it assures that only processes that fulfill the requirements of each other (expressed by urgency conditions) can be composed. Realizability checking imposes a process to respect the requirements of *any* possible environment, which is a much stronger constraint.

To conclude, we have presented a formalism that allows mobile processes to present time-bounded offers and requirements to their environment. This has been done by introducing *clocks* in the π-calculus, making ours the saying that "real-time systems = discrete system + clock variables" [AH94]. The possibility of imposing urgency conditions on certain actions (even taking place on free names) is the mechanism through which a process may ensure itself to retain the *control* of those actions. This is one of the factors that allows one to write non-realizable (*i.e.* environment-constraining) specifications with our language; this is however not peculiar to π^δ, since many process algebras with powerful-enough operators generally lead to unrealizable specifications. We however obtain a strictly finer-grained language, where both the process-algebraic view that a process has no control over its actions and the I/O automata view that a process has control over all its output actions can be expressed.

Chapter 4

A Proof System for π^δ

4.1 Process Equivalence and Abstract Semantics

After proper semantics have been attributed to processes under the form of labeled transition systems, we are left with devising analysis methods for those transition systems. We first notice that labelled transition systems (or their epimorphic unfolding called *synchronization trees* in [Mil83]) associated to processes have been early accused of *over-specifying* the behavior of those processes. This fact being acknowledged by many people (a lot of them belonging to the process-algebraic community), an initially small number of equivalences relations rapidly seeded until a large field of them grew, all differentiated by the testing method they apply to determine when processes with different LTSs have actually undistinguisable behaviors. An exhaustive thesaurus exposing their variety and testing scenarios can be found in [vG01]; the applied taxonomy favors the approach according to which all interesting equivalences belong in the so-called *linear time – branching time spectrum.*

Proposing and studying such a relation is now a mandatory step for all process algebras eager to reach minimal respectability. This section starts with the definition of such an equivalence and continues with pre-requisites for its formal study that will come in the next section. We call our equivalence the *timed late bisimulation*:

Definition 4.1.1 (Timed Late Bisimulation). *A symmetric relation R between processes is a timed late bisimulation if and only if $(t\rho, u\kappa) \in R$ implies:*

- *if $t\rho \xrightarrow{\delta\,\mathcal{S}} t'\rho^{+\delta}$ then there exists u' and $\mathcal{S}'$ such that $\mathcal{S}' \preceq \mathcal{S}$ and $u\kappa \xrightarrow{\delta\,\mathcal{S}'} u'\kappa^{+\delta}$, with $(t'\rho^{+\delta}, u'\kappa^{+\delta}) \in R$;*
- *if $t\rho \xrightarrow{a(b)} t'\rho'$ with $b \notin (\mathrm{fp}(t) \cup \mathrm{fp}(u))$, then there exists u', κ' so that for all names c, $u\kappa \xrightarrow{a(b)} u'\kappa'$ and $(t'[c/b]\rho', u'[c/b]\kappa') \in R$;*
- *if $t\rho \xrightarrow{a(x)} t'\rho'$ with $x \notin (\mathrm{fc}(t) \cup \mathrm{fc}(u))$, then there exists u', κ' so that $\rho'(x) = \kappa'(x)$, $u\kappa \xrightarrow{a(x)} u'\kappa'$, and $(t'\rho', u'\kappa') \in R$;*
- *if $t\rho \xrightarrow{\pi} t'\rho$ for any other π with $\mathrm{bn}(\pi) \cap (\mathrm{fn}(t) \cup \mathrm{fn}(u)) = \emptyset$, then there exists u' so that $u\kappa \xrightarrow{\pi} u'\kappa$ and $(t'\rho, u'\kappa) \in R$.*

The largest such bisimulation is called timed late bisimilarity*, and noted* $t\rho \sim_{tl} u\kappa$*. If the relation is not symmetric, then it defines a preorder called* timed late simulation*, noted* $u\kappa \preceq_{tl} t\rho$*, and* $u\kappa$ *is said to* simulate $t\rho$*.*

This definition stems from both the late bisimilarity of the π-calculus, and the timed bisimilarity as it is defined for various forms of timed automata and processes. Timed bismilarity is not different from the earlier *ground bisimulation* of CCS; it just takes into account the existence of time-passing transitions, and demands that their labeling values match (as it does for other transitions). Yet, our time-passing transitions also feature ready sets: we have to introduce a new condition fixing the relation between ready sets $\mathcal{S}$ and $\mathcal{S}'$. We use the "smaller than" partial order relation $\mathcal{S} \preceq \mathcal{S}'$ defined at the beginning of Section 3.6, stating that $\mathcal{S}'$ must have less offers but more requests than $\mathcal{S}$ for the transition of $u\kappa$ to match the transition of $t\rho$.

On the other side, π-calculi bisimulations are quite different from CCS's ground bisimulation, because they make use of universal quantification over port names: two equivalent processes performing an input should behave identically whatever the name they may receive as argument can be. Hence come the late and early incarnations of bisimilarity, depending on the position of the universal quantification: if it goes first, stating that "*for all the possibly received names there must exists a continuation with matching behavior*", it is the *early* version; if it goes second like above, stating "*there must exist a continuation such that for all the possibly received names the continuation matches*", then it is the *late* incarnation. Obviously, the late equivalence is *stronger* (*i.e.* more discriminating) than the early one.

Our equivalence uses a late quantification over received port names. We however also have to handle received clock names, a necessity inexistent in untimed models. In a similar way to the port name case, we require that process $u\kappa$ exhibits a matching behavior whenever it has received the same value as $t\rho$ did. This is indeed a strong requirement, but it is necessary since the observer may not predict the value received by the process for all the contexts it may be placed in (see the precondition of the axiom PRE of Table 3.2).

The first consequence of the above definition is that timed late bisismilarity, not being preserved by input prefix as its untimed sibling, is not a congruence. Indeed, although

$$([x \leq 1, \bot]\tau)\rho \sim_{tl} ([x \leq 2, \bot]\tau)\rho'$$

for any ρ, ρ' such that $x \in dom(\rho)$, $x \in dom(\rho')$ with $\rho(x) > 2$ and $\rho'(x) > 2$, we have

$$(a(x).[x \leq 1, \bot]\tau)\rho \nsim_{tl} (a(x).[x \leq 2, \bot]\tau)\rho'$$

for any ρ, ρ'. When receiving a clock value, processes must behave the same whatever the received value can be. Yet, when putting those terms in parallel with the term $\nu y\ [1 \leq y < 2, 1 \leq y < 2]\overline{a}(y)$ we certainly obtain different results.

We conjecture that this simple example casts a doubtful shadow on the possibility of having a simple restriction strenghtening timed late bisimilarity to a congruence, as name subtstitution did for strong bisimilarity (leading to *open* bisimilarity [San93]). This doubt holds at least as processes are allowed to communicate clock values to one another when interacting.

We now examine in more details some other after-effects of the above definition, exposing issues induced by having a dense time domain, and following with issues related to name instantiation.

4.1.1 Axiomatizing Dense Time

First of all, a proof system for timed late bisimulation has to cope with the dense time domain, which makes the individual treatment of each couple of processes impossible: to show that P is equivalent to Q, we need to show that for every possible δ, if $P \xrightarrow{\delta} P'$ and $Q \xrightarrow{\delta} Q'$ then P' is equivalent to Q'. Since there is an infinite (non-recursively enumerable) number of possible values for δ, we can not do without a proper *abstraction* for our TTSs, that would associate finitely-many abstract transitions to an infinity of concrete time-passing transitions.

David Dill [Dil89a] has been the first to propose an abstraction in that purpose, and later successfully applied it to timed automata in a collaborative work with Rajeev Alur [AD94]. Those works came in supporting many results for timed automata, allowing to prove the decidability of language emptiness [AD94], bisimulation [Č92] and model checking problems [ACD93]. The divisions in the state space over which time-passing transitions are grouped are called *regions*; for timed automata, the untimed automaton obtained after abstraction is called the *region graph*. Since then, the region abstraction has found ubiquitous applications in real-time system analysis.

Definition 4.1.2 (Region Equivalence). *Given two processes $t\rho$ and $t\rho'$, $ceil(t)$ being the highest natural constant appearing in t, then $t\rho$ and $t\rho'$ belong to the same region if and only if for any clock $x \in fc(t)$:*

- *if $\rho(x) > ceil(t)$ then $\rho'(x) > ceil(t)$, and*
- *if $\rho(x) \leq ceil(t)$ then we have the two properties*
  - *$\lfloor \rho(x) \rfloor = \lfloor \rho'(x) \rfloor$,*
  - *if $\rho(x) - \lfloor \rho(x) \rfloor = 0$ then $\rho'(x) - \lfloor \rho'(x) \rfloor = 0$, and*
- *for all $x, y \in (dom(\rho) \cap dom(\rho'))$ and $p \leq ceil(t)$, $(\rho \vDash x - y \sharp p) \Rightarrow (\rho' \vDash x - y \sharp p)$;*

where $\lfloor \rho(x) \rfloor$ is the integer part of $\rho(x)$.

Intuitively, in a given region two clock valuations ρ and ρ' are identified by forgetting the precise fractional part of the value of each clock, and by not paying attention either to the precise value of a clock if this value exceeds ceil(t) in both ρ and ρ'. To allow a sound abstraction, a region however contains complete information about the order relating the fractional parts of the clocks. It has been shown [ACD93] that the quotient of a timed automaton using the region equivalence still preserves all the branching properties of that automaton regarding the real-time temporal logic TCTL. It has also been shown by Tripakis and Yovine that the region graph of a timed automaton is actually the quotient of this automaton by a (very) *strong time-abstracting bisimulation* [TY01].

We now present an adaptation of time-abstracting bisimulation to our semantic setting. It resembles timed bisimulation, but delays over time-passing transitions are not required to match, and port name instantiation is ignored. The relation also abstracts from the exact clock values that may be received from the environment of the process: the second case, the one dealing with transitions $t\rho \xrightarrow{a(x)} t'\rho'$, obliges only $u\kappa$ to have a transition $u\kappa \xrightarrow{a(x)} u'\kappa'$, ignoring which value is actually received. The same case similarly identifies all clock outputs, here also featuring no requirement on the actual output value. Those two latter points seem specific to our setting, no other model that we know of allowing to transmit clock values during communication.

Definition 4.1.3 (Time-Abstracting ground Bisimulation). *A symmetric relation R between processes is a time-abstracting ground bisimulation if and only if $(t\rho, u\kappa) \in R$ implies:*

- *if $t\rho \xrightarrow{\delta\ \mathcal{S}} t'\rho^{+\delta}$ there exists u', δ' so that $u\kappa \xrightarrow{\delta'\ \mathcal{S}} u'\kappa^{+\delta'}$ and $(t'\rho^{+\delta}, u'\kappa^{+\delta}) \in R$;*
- *if $t\rho \xrightarrow{\lambda} t'\rho'$ for any other λ, then there exists u' so that $u\kappa \xrightarrow{\lambda} u'\kappa'$ and $(t'\rho', u'\kappa') \in R$.*

The largest such bisimulation is called time-abstracting ground bisimilarity*, and noted $t\rho \sim_{tag} u\kappa$. If the relation is not symmetric, then it defines a preorder called* time-abstracting ground simulation*, noted $u\kappa \preceq_{tag} t\rho$.*

The equivalence classes obtained through time-abstracting (ground) bisimulation are called *zones*. A zone is a union of regions, hence providing a more compact way of representing the state space of a process. The symbolic semantics that we present in Section 4.1.4 is applied to zones, that are obtained by computing the quotient of the state space of concrete terms through time-abtracting ground bisimulation. However, before we elaborate more on this and give further developments on practical issues about zones in Section 4.1.3, we introduce another important problem.

4.1.2 Axiomatizing Name Instantiation

We have seen that π-calculi bisimulations require a universal quantification over names to deal with input prefixes. Since there are (countably) infinitely-many names, proving the equivalence of two processes here also requires proving an infinite number of sub-goals, one for each possible name.

Fortunately, in this case as well many solutions have been proposed: starting by Milner, Parrow and Walker [MPW92] with their *name distinction sets*, the axiomatization of *open* bisimulation by Sangiorgi [San93], the explicit substitutions of Ferrari, Montanari and Quaglia [FMQ94], and the applications of *symbolic bisimulations* [HL95] to the π-calculus by Lin and Hennessy [Lin94], Boreale and De Nicola [BD94]. A different but related proposal concerns *history-dependent automata* [MP98], automata that are expressive enough to simulate all π-calculus terms and that have been given an axiomatization for bisimulation [MP95].

In a nutshell, all those contributions solve the problem using symbolic transition systems where transitions are equipped with additional information about which

names are syntactically equal and which names are syntactically distinct. As an example, the matching operator $[a = b]t$ allows a transition for t only if this transition agrees on the equality of names a and b. Any transition assuming otherwise cannot occur.

Another exemplary case is the input prefix $a(b).\,t$: after the transition the placeholder b may be equal to one of the free names of the continuation t, or it may be a "new" name, different from all other free names in t. Hence, the cardinal of $\mathrm{fc}(t)$ being $|\mathrm{fc}(t)|$, there should be $|\mathrm{fc}(t)|+1$ separate discrete transitions leaving $a(b).\,t$ to as many equivalence classes addressing a different case. For instance, the process t reached after the symbolic transition $a(b).\,t \xrightarrow{c \text{ is new in } t} t$ is an equivalence class representing an infinite number of concrete processes for which a new name $c, d, e, \cdots$ different from any name in $\mathrm{fc}(t)$ has been received through a concrete transition; since c was not free in t, they actually all behave in the same way, though.

We start more technical developments by giving some ancillary but indispensable notations for manipulating zones and regions through clock constraints, and symbolic name sets through matching constraints.

4.1.3 Manipulating Zones, Regions, and Matching Constraints

The delimitation on clock values for zones and regions will be represented using clock constraints, which will be typically named ζ or ξ. Those contraints use the same syntax as the general constraints of Section 3.4:

$$\zeta ::= \zeta \wedge \zeta \mid \zeta \vee \zeta \mid \neg\, \zeta \mid x \sharp q \mid x - y \sharp q$$

where $\sharp \in \{<, =, >\}$ and $q \in \mathbb{Q}_+$. True and false values are noted as above $\top$ and $\bot$, respectively. Zone constraint and region constraint satisfaction is constraint satisfaction (see again Section 3.4).

Regions contraints are represented as clock constraints possessing a particular format. A clock constraint ζ forms a region $t\,\zeta$ with process term t over clocks $\mathrm{fc}(t) = x_1, \cdots, x_k$ if it respects the *minimality* and the *proper ordering* properties:

- for each $i \in \{1, \ldots, k\}$, either $x_i = p_i$ or $p_i < x_i < p_i + 1$, or $x > \mathit{ceil}(x, t)$, and
- for each pair $i, j \in \{1, \ldots, k\}$ with $i \neq j$, then either $x_i - x_j = p$ or $x_i - x_j < p$ with $p \leq \mathit{ceil}(x_i, t)$, or $x_j - x_i < p$ with $p \leq \mathit{ceil}(x_j, t)$.

For verification purposes, it has been however found rapidly that regions are too discriminating abstractions: processes belonging to different regions often behave in the same way. To symbolically represent the state space using *zones* has thus been proposed, by Henzinger, Nicollin, Sifakis and Yovine [HNSY92]. Eventually, zones were characterized as representatives of equivalence classes obtained through time-abstracting bisimulation [TY01].

Like a region, a zone is a couple $t\,\zeta$ where t is a process term, and ζ is a clock constraint over the free clocks $x_1, \cdots, x_k$ of t. We take ζ to be a disjunction of *contiguous* region constraints; ζ is also assumed to be tagged with a *ceiling* value for each $x_1, \cdots, x_k$. This ceiling value will be noted $\mathit{ceil}(x, \zeta)$; yet, for the sake of simplicity, we shall omit to apply the ceiling tag when writing zone constraints, and

we shall allow to freely examine the tag when convenient, on the grounds that it has remained implicitly present in the scriptures. Unless otherwise mentioned, all operators defined on zone constraints preserve ceiling tags, while predicates on zone constraints ignore them.

Zones are in general not minimal, and they do not repect proper ordering either. However, any zone is composed by a finite number of regions [TY01]. It has furthermore been shown in [HNSY92] that zones can be maintained so that all TCTL properties of a timed automaton are decidable on its zone-enabled abstraction, without having to explicitly compute the entire set of regions reachable by the automaton.

We now define some elementary operations on zone constraints, that roughly correspond to the ones defined on clock valuations in Section 3.2. An illustration of the effect of those operations on 2-polyhedra is given in Figure 4.1, that we borrowed from [TY01].

For each operation, we give a definition on logic grounds (*i.e.* on constraints), and its correspondence in set-theoretic terms. We overload all the introduced operators so that we can apply them to both constraints and their semantic clock valuation sets uniformly.

In the following definitions, ζ is assumed to be in *region-disjunctive normal form*, that is of the form $\zeta \triangleq \zeta_{\wedge 1} \vee \cdots \vee \zeta_{\wedge k}$ where each $\zeta_{\wedge i}$ is a region constraint. It is not a limitation, since any constraint can be rewritten in this form. For convenience, each operation is hence defined at first on region constraints, then extented to constraints in region-disjunctive normal form. Operations defined on region constraints do not yield region constraints, but plain general constraints. By taking implication to form a preorder over constraints, we say that a constraint ξ is *weaker* than another constraint ξ' if and only if $\xi' \Rightarrow \xi$.

When the restriction from a clock x, noted $\zeta^{\backslash x}$, is applied to a region constraint $\zeta_\wedge$, the result $\zeta_\wedge^{\backslash x}$ is the weakest constraint satisfying for any clocks $y \neq x$ and $z \neq x$, and for any constant p:

- if $\zeta_\wedge \Rightarrow y \sharp p$, then $\zeta_\wedge^{\backslash x} \Rightarrow y \sharp p$ and,
- if $\zeta_\wedge \Rightarrow y - z \sharp p$, then $\zeta_\wedge^{\backslash x} \Rightarrow y - z \sharp p$.

For a general constraint we obtain naturally: $\zeta^{\backslash x} \triangleq \zeta_{\wedge 1}^{\backslash x} \vee \cdots \vee \zeta_{\wedge k}^{\backslash x}$. The corresponding set-theoretic characterization is that $\rho \in [\![\zeta^{\backslash x}]\!]$ if and only if $\exists \rho' \in [\![\zeta]\!]. \forall y \in (clocks(\zeta) \backslash \{x\}). \rho'(y) = \rho(y)$. We hence trivially have $\rho \in [\![\zeta]\!] \Rightarrow \rho^{\backslash x} \in [\![\zeta^{\backslash x}]\!]$; the converse is false, though. The restriction operator can be generalized to sets of clocks X, with $\zeta^{\backslash X}$ defined inductively by $\zeta^{\backslash \emptyset} \triangleq \zeta$ and $\zeta^{\backslash X} \triangleq (\zeta^{\backslash X'})^{\backslash x}$ for some $x \in X$ with $X' \triangleq X \backslash \{x\}$. The converse operation $\zeta^{/X}$ is defined by $\zeta^{\backslash (clocks(\zeta) \backslash X)}$.

The constraint $\zeta^{\downarrow x}$ is obtained from ζ by reseting clock x. As for clock valuations, the clock x has to be new in constraint ζ: $x \notin \text{clocks}(\zeta)$. For a region $\zeta_\wedge$, for any clock $y \neq x$ and constant p, $\zeta_\wedge^{\downarrow x}$ is the weakest conjunctive constraint satisfying:

- $\zeta_\wedge^{\downarrow x} \Rightarrow \zeta_\wedge$,
- if $\zeta_\wedge \Rightarrow (y = p)$ then $\zeta_\wedge^{\downarrow x} \Rightarrow (y - x = p)$,
- if $\zeta_\wedge \Rightarrow (y > p)$ then $\zeta_\wedge^{\downarrow x} \Rightarrow (y - x > p)$, and

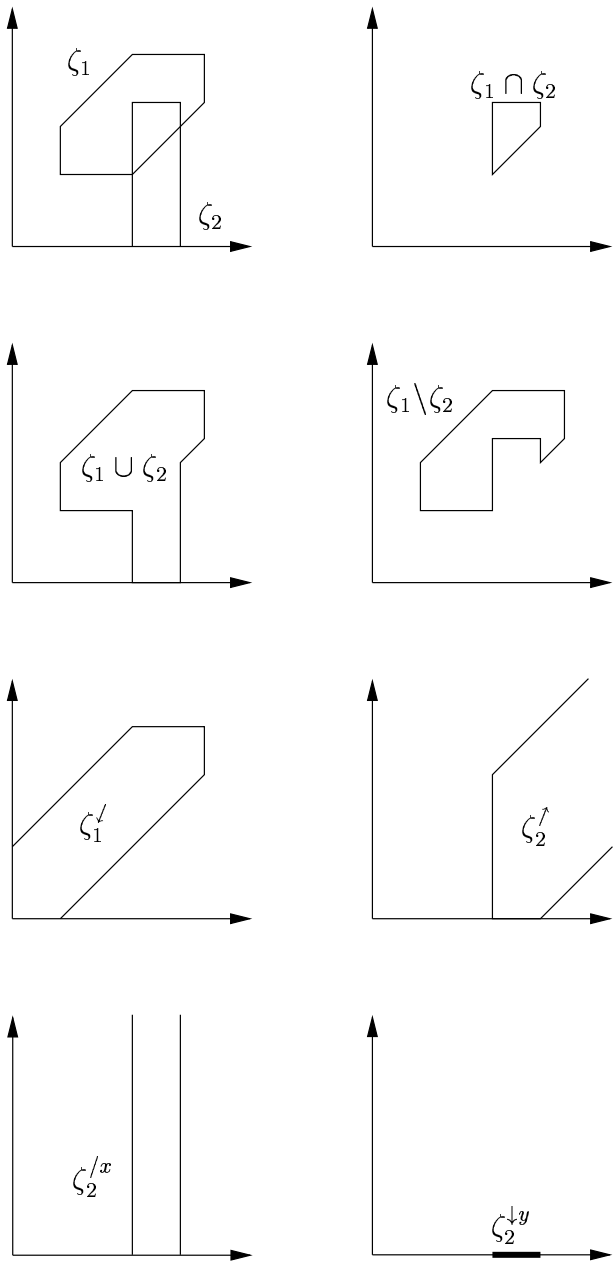

Figure 4.1: Operations on 2-Polyhedra

- $\zeta_\wedge^{\downarrow x} \Rightarrow (x = 0)$.

Then, as before $\zeta^{\downarrow x} \triangleq \zeta_{\wedge 1}^{\downarrow x} \vee \cdots \vee \zeta_{\wedge k}^{\downarrow x}$. The set-theoretic characterization of this operation is straightforward: $[\![\zeta^{\downarrow x}]\!] = \{\rho^{\downarrow x} \mid \rho \in [\![\zeta]\!]\}$.

The upward and downward opening operations are true of all the valuations reachable by respectively letting time progress or regress at will from one valuation satisfying the present constraint. Thus the upward (resp. downward) opening $\zeta^\nearrow$ (resp. $\zeta^\swarrow$) of ζ is obtained by removing the upper bounds (resp. lower bounds) constraints on all the clocks of ζ, and by fixing the order respected by their fractional parts. Indeed, if the values of the clocks may progress or regress arbitrarily, all of them should do it at the same rate, the order existing among their respective fractional parts remaining unchanged. Thus, for a region constraint $\zeta_\wedge$, for any clocks x, y, and constants p, q, then $\zeta_\wedge^\nearrow$ is the weakest constraint satisfying:

- $\zeta_\wedge \Rightarrow \zeta_\wedge^\nearrow$,
- if $\zeta_\wedge \Rightarrow (x = p)$, then $\zeta_\wedge^\nearrow \Rightarrow (x \geq p)$,
- if $\zeta_\wedge \Rightarrow (x - y < p)$, then $\zeta_\wedge^\nearrow \Rightarrow (x - y < p) \wedge (x - y > p - 1)$,
- if $\zeta_\wedge \Rightarrow (x = p \wedge y = q)$ then $\zeta_\wedge^\nearrow \Rightarrow x - y = p - q$.

Then again, $\zeta^\nearrow \triangleq \zeta_{\wedge 1}^\nearrow \vee \cdots \vee \zeta_{\wedge k}^\nearrow$. The definition of $\zeta^\swarrow$ is identical to the one of $\zeta^\nearrow$, except "$>$" is replaced by "$<$", and "$\geq$" is replaced by "$\leq$". The set-theoretic characterizations are more direct: $[\![\zeta^\nearrow]\!] = \{\rho' \mid \exists \rho, \delta.\ \rho \in [\![\zeta]\!] \wedge \rho' = \rho^{+\delta}\}$ and $[\![\zeta^\swarrow]\!] = \{\rho \mid \exists \rho', \delta.\ \rho' \in [\![\zeta]\!] \wedge \rho' = \rho^{+\delta}\}$. We remark that $\top^\nearrow = \top^\swarrow = \top$ and $\bot^\nearrow = \bot^\swarrow = \bot$.

Finally, we shall write that a set $\zeta_1, \cdots, \zeta_k$ is a *partition* of a zone constraint ζ (or equivalently that $\zeta_1, \cdots, \zeta_k$ is a ζ-partition) if and only if $\zeta \Leftrightarrow (\zeta_1 \vee \cdots \vee \zeta_k)$.

We now define the (simpler) operations on matching constraints. Typical name matching constraints range over μ, η, θ and their decorated variants. For any port names a and b matching constraints respect the syntax:

$$\mu ::= \ a = b \mid a \neq b \mid \mu \wedge \mu \ .$$

Our matching constraints are thus strictly conjunctive. The set of port names appearing in a matching constraint μ is noted $ports(\mu)$. We note the true and false value respectively $\top$ and $\bot$ as above, defining them for example as $\top \triangleq x = x$ and $\bot \triangleq x \neq x$.

As we lack negation and disjunction in their definition, we have to define an implication relation over matching constraints in an ad-hoc, axiomatic fashion. If we note $\mu \Leftrightarrow \eta$ if and only if $\mu \Rightarrow \eta \wedge \eta \Rightarrow \mu$, then it is defined as the smallest

relation closed under the following rules for any matching constraints μ, η, θ:

$(\mu \wedge \eta) \Leftrightarrow (\eta \wedge \mu)$	commutativity
$(\mu \wedge \eta) \wedge \theta \Leftrightarrow \mu \wedge (\eta \wedge \theta)$	associativity
$(\mu \Rightarrow \eta) \wedge (\eta \Rightarrow \theta) \Rightarrow (\mu \Rightarrow \theta)$	transitivity
$(\mu \Rightarrow \eta) \wedge (\mu \Rightarrow \theta) \Rightarrow (\mu \Rightarrow \eta \wedge \theta)$	consequent
$\mu \wedge \eta \Rightarrow \mu$	weakening
$\mu \Rightarrow \top,\ \bot \Rightarrow \mu$	$\top$ and $\bot$ laws
$a = b \Rightarrow b = a,\ a \neq b \Rightarrow b \neq a$	
$a = b \wedge a = c \Rightarrow b = c$	
$a = b \wedge a \neq c \Rightarrow b \neq c$	

We remark that reflexivity can be inferred from the previous system: using consequent and weakening on $\mu \wedge \top \Rightarrow \mu$, we obtain $(\mu \wedge \top) \wedge (\mu \wedge \top) \Rightarrow (\mu \wedge \mu)$, hence $\mu \wedge \mu \Rightarrow \mu$, and by transitivity $\mu \Rightarrow \mu$.

We generalize the application of substitutions to name matching constraints: with $\alpha \triangleq [\tilde{n}/\tilde{m}]$, $\mu\alpha$ is the matching constraint obtained after simultaneous substitution of each name m_i in $\tilde{m}$ by the corresponding n_i is $\tilde{n}$. We say that a substitution α *is consistent with*, *respects*, or *satisfies* a matching constraint μ, written $\alpha \vDash \mu$, if for all $a, b \in ports(\mu)$:

$$((\mu \Rightarrow a = b) \Rightarrow (\alpha(a) = \alpha(b))) \wedge ((\mu \Rightarrow a \neq b) \Rightarrow (\alpha(a) \neq \alpha(b)))\ .$$

It is simple to show that $\mu \Rightarrow \eta$ if and only if for all substitutions α, $\alpha \vDash \mu \Rightarrow \alpha \vDash \eta$.

Each name matching constraint μ allows the definition of an indexed equivalence relation $\equiv^{\mu}$ over transition labels. It is defined as the smallest relation closed under the following rules:

$\tau \equiv^{\mu} \tau$		
$\overline{a}x \equiv^{\mu} \overline{b}x$	if	$\mu \Rightarrow a = b$
$\overline{a}c \equiv^{\mu} \overline{b}d$	if	$\mu \Rightarrow (a = b \wedge c = d)$
$\overline{a}(c) \equiv^{\mu} \overline{b}(d)$	if	$\mu \Rightarrow a = b$
$a(m) \equiv^{\mu} b(n)$	if	$\mu \Rightarrow a = b$

We say that a matching constraint μ belongs to the *maximally consistent extension* of a matching constraint η over a set of port names $\mathcal{A} = \{a_1, \cdots, a_k\}$, written $\mu \in MCE_{\mathcal{A}}(\eta)$, if $\mu \Rightarrow \eta$, and for any elementary constraint $a = b$ with $\{a, b\} \subseteq (clocks(\eta) \cup \mathcal{A})$, then either $\mu \Rightarrow a = b$ or $\mu \Rightarrow a \neq b$.

The definition of the equivalence that we will use on our symbolic TTSs makes a crucial use of partitions and extensions: it basically requires that the transitions of the compared terms match for all the zones of a zone constraint partition and some maximally consistent name matching constraint. The existence of such partitions and maximally consistent name matching constraint is decidable over finite terms, since the number of regions (and hence partitions) and the number of maximally consistent name constraints are finite.

4.1.4 Abstract Labeled Transition Systems

Under the lights of Sections 4.1.1 and 4.1.2, our abstract labeled transition systems are the result of two orthogonal abstractions: the one dealing with time, and the other dealing with name instantiation.

We give a *symbolic operational semantics* [HL95] to zones: name-related abstraction information is bound to transitions, while time-related abstraction information is bound to process terms. Although it would be possible to place the information about names also on processes, we think that proceeding this way encourages a clear separation of concerns between the two abstractions. To be complete, one may ask the question: *why do you not rather place time-related* and *name-related information on transitions*? This question turns out to be interesting, and we shall give it a proper answer in Section 4.3.

As a result of the previous considerations, our symbolic transitions may carry two sorts of labels:

$$\lambda_s ::= \pi,\ \mu \mid \langle \mathcal{R} : \mathcal{O} \rangle$$

where π is an action as defined earlier, μ is a name matching constraint, and $\langle \mathcal{R} : \mathcal{O} \rangle$ is a ready set for time-passing actions. Hence, discrete transitions now carry matching constraints for name abstraction, while time-passing transitions logically do not bear any delay, having undergone time-abstraction. The fundamental property that we try to obtain with our abtract transition systems, and that we shall indeed prove later, is that an abtract process $t\,\zeta$ may perform a transition only if any concrete process $t\rho$ with $\rho \vDash \zeta$ can perform a corresponding concrete transition. The correspondence for discrete transitions $t\zeta \xrightarrow{\pi,\mu} t'\,\zeta'$ is judged modulo equalities deducible from the name matching constraint μ, authorizing $t\rho \xrightarrow{\pi'} t'\rho'$ when $\pi \equiv^{\mu} \pi'$. Time-passing transition correspondence is judged simply by ready-set correspondence: $t\zeta \xrightarrow{\mathcal{S}} t'\,\zeta'$ only if $t\rho \xrightarrow{\delta\mathcal{S}} t'\rho^{+\delta}$ and $\zeta' \vDash \rho^{+\delta}$.

We present the symbolic operational semantics in the same style as the concrete one, starting by the definition of a "ground" structural congruence relation $\equiv^g$ over processes. This relation is nearly identical to the structural congruence of Section 3.6. The first noticable difference is that we remove the rules concerning the matching operator, so that it can be dealt with explicitly, in a symbolic fashion. To simplify the reduction rules, we add one axiom allowing us to consider only terms where the selection condition for a prefix action cannot stay true after the urgency condition on the same prefix became false. The rules are given on Table 4.1.

As in the concrete case, we then extend the ground structural congruence $\equiv^g$ by allowing the identification of free clock names that yield the same value. Formally, for two terms t, u and two constraints ζ, ξ with $\mathrm{fc}(t) \subseteq \mathrm{clocks}(\zeta)$ and $\mathrm{fc}(u) \subseteq \mathrm{clocks}(\xi)$, if we take $\mathrm{fc}(t)$ and $\mathrm{fc}(u)$ to be arbitrarily ordered respectively as $x_1, x_2, \cdots, x_k$ and $y_1, y_2, \cdots, y_k$, then

$$t\,\zeta \equiv u\,\xi \Leftrightarrow \left\{ \begin{array}{l} t\,\zeta \equiv^g u\,\xi \ \wedge \\ \exists z_1, z_2, \cdots, z_k. \ \left\{ \begin{array}{l} set(\tilde{z}) \cap (set(\tilde{x}) \cup set(\tilde{y})) = \emptyset \ \wedge \\ \zeta^{/\mathrm{fc}(t)}[\tilde{z}/\tilde{x}] \Leftrightarrow \xi^{/\mathrm{fc}(u)}[\tilde{z}/\tilde{y}]\,. \end{array} \right. \end{array} \right.$$

with for all $i, j \in \{1..k\}$, $i \neq j$, z_i, z_j pairwise different. There again De Bruijn indices may be used to efficiently find such a mapping $z_1, z_2, \cdots, z_k$.

$$(t+0)\,\zeta \equiv^g t\,\zeta \quad (t \mid 0)\,\zeta \equiv^g t\,\zeta \quad (\nu n\ 0)\,\zeta \equiv^g 0\ \zeta \quad ([a=b]0)\,\zeta \equiv^g 0\ \zeta$$
$$(t+u)\,\zeta \equiv^g (u+t)\,\zeta \quad ((t+u)+v)\,\zeta \equiv^g (t+(u+v))\,\zeta$$
$$(t \mid u)\,\zeta \equiv^g (u \mid t)\,\zeta \quad ((t \mid u) \mid v)\,\zeta \equiv^g (t \mid (u \mid v))\,\zeta$$
$$(\nu n\ t)\,\zeta \equiv^g (\nu m\ u)\,\zeta \ \textbf{if}\ t[o/n] = u[o/m]\ \textbf{for some fresh name}\ o$$
$$([\sigma, \upsilon]\pi.\, t)\,\zeta \equiv^g ([\sigma \wedge \upsilon^{\checkmark}, \upsilon]\pi.\, t + [\sigma \backslash \upsilon^{\checkmark}, \bot]\pi.\, t)\,\zeta$$

Table 4.1: The Symbolic Structural Congruence

$$\textsc{Pre}\ \frac{\zeta \Rightarrow \sigma \quad \zeta' = reach(\zeta, \pi)}{([\sigma, \upsilon]\pi.\, t)\,\zeta \xrightarrow{\pi, \top} t\,\zeta'} \qquad \textsc{Com}\ \frac{t\,\zeta \xrightarrow{a(m), \mu} t'\,\zeta' \quad u\,\xi \xrightarrow{\overline{b}\,n, \eta} u'\,\xi'}{(t \mid u)(\zeta \wedge \xi) \xrightarrow{\tau, \mu\wedge\eta\wedge\theta} (t'[n/m] \mid u)(\zeta'^{\backslash m} \wedge \xi')}\ \blacktriangle$$

$$\textsc{Open}\ \frac{t\,\zeta \xrightarrow{\overline{a}\,b, \mu} t'\,\zeta'}{(\nu b\ t)\,\zeta \xrightarrow{\overline{a}(b), \mu} t'\,\zeta'}\ \nabla \qquad \textsc{Close}\ \frac{t\,\zeta \xrightarrow{a(c), \mu} t'\,\zeta' \quad u\,\xi \xrightarrow{\overline{b}(c), \eta} u'\,\xi'}{(t \mid u)(\zeta \wedge \xi) \xrightarrow{\tau, \mu\wedge\eta\wedge\theta} ((c)(t' \mid u'))(\zeta' \wedge \xi')}\ \blacktriangleright$$

$$\textsc{Res}\ \frac{t\,\zeta \xrightarrow{\pi, \mu} t'\,\zeta'}{(\nu a\ t)\,\zeta \xrightarrow{\pi, \mu} (\nu a\ t')\,\zeta'}\ \blacktriangledown \qquad \textsc{Reset}\ \frac{t\,\zeta^{\downarrow x} \xrightarrow{\lambda_s} t'\,\zeta'}{(\nu x\ t)\,\zeta \xrightarrow{\lambda_s} t'\,\zeta'}\ \Box$$

$$\textsc{Sum}\ \frac{t\,\zeta \xrightarrow{\pi, \mu} t'\,\zeta'}{(t+u)\,\zeta \xrightarrow{\pi, \mu} t'\,\zeta'}\ \Diamond \qquad \textsc{Par}\ \frac{t\,\zeta \xrightarrow{\pi, \mu} t'\,\zeta'}{(t \mid u)\,\zeta \xrightarrow{\pi, \mu} (t' \mid u)\,\zeta'}\ \blacklozenge$$

$$\textsc{Id}\ \frac{t\,[\tilde{n}/\tilde{m}]\,\zeta \xrightarrow{\lambda_s} t'\,\zeta'}{(A(\tilde{n}))\,\zeta \xrightarrow{\lambda_s} t'\,\zeta'}\ A(\tilde{m}) \triangleq t \qquad \textsc{Match}\ \frac{t\,\zeta \xrightarrow{\pi, \mu} t'\,\zeta'}{([a=b]t)\,\zeta \xrightarrow{\pi, \mu\wedge\theta} t'\,\zeta'}\ \Delta$$

$$\textsc{Conv}\ \frac{u\xi \equiv t\zeta \quad t\zeta \xrightarrow{\lambda_s} t'\zeta' \quad u'\xi' \equiv t'\zeta'}{u\xi \xrightarrow{\lambda_s} u'\xi'}$$

$$\Delta \triangleq\ \theta \Leftrightarrow \begin{cases} a=b \text{ if } a \neq b, \\ \top \text{ otherwise.} \end{cases} \qquad \blacktriangle \triangleq \Delta \wedge ((\zeta \wedge \xi) \not\Leftrightarrow \bot) \wedge ((\zeta'^{\backslash m} \wedge \xi') \not\Leftrightarrow \bot)$$
$$\blacktriangleright \triangleq \Delta \wedge ((\zeta \wedge \xi) \not\Leftrightarrow \bot) \wedge ((\zeta' \wedge \xi') \not\Leftrightarrow \bot)$$
$$\nabla \triangleq b \notin (\mathrm{ports}(\mu) \cup \{a\}) \qquad \blacktriangledown \triangleq a \notin (n(\mu) \cup n(\pi))$$
$$\Diamond \triangleq \mathrm{clocks}(\zeta) \supseteq (\mathrm{fc}(t) \cup \mathrm{fc}(u)) \qquad \blacklozenge \triangleq \Diamond \wedge ((\mathrm{bn}(\pi) \cap \mathrm{fn}(u)) = \emptyset)$$
$$\Box \triangleq (x \notin clocks(\zeta)) \wedge (ceil(x, \zeta^{\downarrow x}) = ceil(x, t))$$

Table 4.2: Symbolic Transitional Semantics for π^{δ} : part 1

The rules for abstract discrete transitions, reported in Table 4.2, are very similar to the concrete rules of Table 3.2, but they have been adapted according to the principles enounced in [Lin94]. The first rule PRE for input prefixes is the only axiom for discrete transitions. It requires that all the concrete terms in the current zone may trigger the prefix, hence $\zeta \Rightarrow \sigma$; it imposes no constraint on name equality. The set of reachable clock constraints $reach(\zeta, \pi)$ from a constaint ζ through transition π is defined by: $reach(\zeta, \pi) \triangleq (\zeta \wedge x \geq 0)$ if $\pi = a(x)$ for some a, and $reach(\zeta, \pi) \triangleq \zeta$ otherwise. By comparison to the concrete case, one may see that those abstract transition systems are now *image finite*, while our concrete transitions systems were not. An infinite number of transitions obtained through rule PRE are now abstracted to one sole transition obtained through rule PRE.

The rules COM and CLOSE have been modified to require the equality on the subject of the interaction to be accomplished. They allow the conjunctive splitting of the zone constraint into ζ and ξ, at the condition that $\zeta \wedge \xi$ is not an antilogy. The rule OPEN states that a name b can be hidden if a transition with the same b not hidden can be deduced when considering b different from all the other names in the term t. This is written by using the predicate $\text{unique}(b, \{a_1, \ldots, a_n\})$ defined to be $(b \neq a_1) \wedge \cdots \wedge (b \neq a_n)$. Once b is hidden, this constraints logically disappears.

Restriction and reset rules are identical to the concrete case, although this time of course, the reset rule allows transitions carrying λ_s labels. The rules for choice also allow similar behaviors to the concrete case, without introducing any constraint on port names. More interesting is the rule for name matching, that forces equality on port names a and b, as described earlier in Section 4.1.2.

The rules for time progression in Tables 4.3 and 4.4 are quite different, and somewhat more intricate than in the concrete case. This is because we need to cope with two hindrances: the dense nature of time, and the failure of our models to withstand the *time continuity* property, as seen in Section 3.8. The general idea behind the rules in Table 4.3 is therefore to accommodate the variations that may affect the ready sets placed over transitions: a prefix transition may be either impossible, selected but not urgent (meaning its name will be in the offer set), or urgent. This introduces three *frontiers* of *time discontinuity* that limitate the time progression a process may accomplish in one step. A process may let time advance up to the next reachable frontier, but it has to stop there mandatorily, the set of time-passing transitions enabled from the frontier bearing a different ready set from the ones before. This obligation has the virtue of ensuring the proper execution of urgent interactions: stopping at the frontier allows them to impeach further time-passing transitions and to execute right at the moment when they become urgent. Errors are dealt with according the same spirit in Table 4.4. In the same table can be found the rules for restriction, choice, and parallel composition.

The rules TSEL,TURG and TELA in Table 4.3 allow a given zone to reach the next possible frontier, if it exists. The rules TIDL, TUNBS and TUNBU deal with *unbounded* zones: zones in which time may pass at will without the concrete processes in the zone crossing any frontier of time discontinuity. The rule TREG "complements" the six rules above it, stating that if a zone ζ' (that zone being either an unbounded zone or a frontier of time discontinuity) is reachable, then all the zones between the present one and the given ζ' are themselves reachable (this corresponds

to the *interval trajectory* property of Section 3.8).

The rule TMISS in Table 4.4 deals with obligations: a process produces an error if its environment required an interaction that will not be produced under the specified timeliness conditions. The rule TREQ is similar to TREG, but it is specially needed for the case where an urgent interaction is required to happen, TREG being ineffective in this case because rule TMISS does not lead to an executable term but to *Error*. The rule TERR is used to block the execution with explicit errors, allowing time to progress until the condition υ becomes true, the process then becoming deadlocked. TNERR lets time elapse forever, an error never being reached.

Finally, the rules TSUM and TPAR allow time progression in choice or composition processes, at the condition that the same time elapses in both subterms. The rule TRES for restriction is almost identical to the concrete TRES rule.

$$\textsc{Tsel}\ \frac{((\zeta \wedge \sigma^{\nearrow}) \Leftrightarrow \bot) \wedge (\zeta^{\nearrow\swarrow} \Rightarrow \sigma^{\nearrow\swarrow})}{([\sigma,\upsilon]\pi.\,t)\,\zeta \xrightarrow{\langle\rangle} ([\sigma,\upsilon]\pi.\,t)(\sigma \wedge \zeta^{\nearrow})^{=\swarrow}}$$

$$\textsc{Turg}\ \frac{(\zeta \Rightarrow (\sigma \backslash \upsilon^{\nearrow})) \wedge (\zeta^{\nearrow\swarrow} \Rightarrow \upsilon^{\nearrow\swarrow})}{([\sigma,\upsilon]\pi.\,t)\,\zeta \xrightarrow{\langle\emptyset:s(\pi)\rangle} ([\sigma,\upsilon]\pi.\,t)(\upsilon \wedge \zeta^{\nearrow})^{=\swarrow}}$$

$$\textsc{Tela}\ \frac{(\zeta \Rightarrow (\sigma \backslash \upsilon^{\nearrow})) \wedge (\sigma^{\nearrow} \not\Leftrightarrow \sigma) \wedge ((\upsilon \wedge \zeta^{\nearrow}) \Leftrightarrow \bot)}{([\sigma,\upsilon]\pi.\,t)\,\zeta \xrightarrow{\langle\emptyset:s(\pi)\rangle} ([\sigma,\upsilon]\pi.\,t)(\sigma \wedge \zeta^{\nearrow})^{\nearrow=}}$$

$$\textsc{Tidl}\ \frac{((\zeta^{\nearrow} \backslash \zeta^{=\swarrow}) \wedge \sigma^{\swarrow}) \Leftrightarrow \bot}{([\sigma,\upsilon]\pi.\,t)\,\zeta \xrightarrow{\langle\rangle} ([\sigma,\upsilon]\pi.\,t)\,\zeta^{ceil}}$$

$$\textsc{Tunbs}\ \frac{(\zeta \Rightarrow \sigma) \wedge (\sigma^{\nearrow} \Leftrightarrow \sigma) \wedge ((\upsilon \wedge \zeta^{\nearrow}) \Leftrightarrow \bot)}{([\sigma,\upsilon]\pi.\,t)\,\zeta \xrightarrow{\langle\emptyset:s(\pi)\rangle} ([\sigma,\upsilon]\pi.\,t)\zeta^{ceil}}$$

$$\textsc{Tunbu}\ \frac{(\zeta \Rightarrow \upsilon) \wedge (\upsilon^{\nearrow} \Leftrightarrow \upsilon)}{([\sigma,\upsilon]\pi.\,t)\,\zeta \xrightarrow{\langle s(\pi):\emptyset\rangle} ([\sigma,\upsilon]\pi.\,t)\,\zeta^{ceil}}$$

$$\textsc{Treg}\ \frac{([\sigma,\upsilon]\pi.\,t)\,\zeta \xrightarrow{\mathcal{S}} ([\sigma,\upsilon]\pi.\,t)\,\zeta' \quad (\xi \Rightarrow ((\zeta^{\nearrow} \backslash \zeta^{=\swarrow}) \wedge \zeta'^{\swarrow})) \wedge (\xi^{\nearrow\swarrow} \Leftrightarrow \zeta^{\nearrow\swarrow})}{([\sigma,\upsilon]\pi.\,t)\,\zeta \xrightarrow{\mathcal{S}} ([\sigma,\upsilon]\pi.\,t)\,\xi}$$

Table 4.3: Symbolic Transitional Semantics for π^{δ} : part 2

Before we examine the rules of Tables 4.3 and 4.4 in more detail, we have to define some complementary operations on zones that are used in them. Consider a region constraint $\zeta_{\wedge}$. A clock $x \in \mathrm{clocks}(\zeta_{\wedge})$ belongs to the set of upward most constraining clocks $\mathcal{C}^{\nearrow}$ if and only if $\zeta_{\wedge} \Rightarrow x < p$ for some p and, for any clock $y \neq x$ and constant q, if $\zeta_{\wedge} \Rightarrow y < q$, then $\zeta_{\wedge} \Rightarrow x - y \geq p - q$. Symmetrically, A x belongs to the set

$$\textsc{Tmiss}\ \frac{(\zeta \Rightarrow \upsilon) \wedge (\upsilon^{\nearrow} \not\Leftrightarrow \upsilon)}{([\sigma,\upsilon]\pi^\star.\, t)\,\zeta \xrightarrow{\langle s(\pi^\star):\emptyset\rangle} Error((\upsilon \wedge \zeta^{\nearrow})^{\nearrow^{=}})}$$

$$\textsc{Treq}\ \frac{(\zeta \Rightarrow \upsilon) \wedge (\upsilon^{\nearrow} \not\Leftrightarrow \upsilon) \wedge (\xi \Rightarrow (\upsilon \wedge (\zeta^{\nearrow} \backslash \zeta^{=\swarrow}))) \wedge (\xi^{\nearrow\swarrow} \Leftrightarrow \zeta^{\nearrow\swarrow})}{([\sigma,\upsilon]\pi.\, t)\,\zeta \xrightarrow{\langle s(\pi):\emptyset\rangle} ([\sigma,\upsilon]\pi.\, t)\xi}$$

$$\textsc{Terr}\ \frac{((\zeta \wedge \upsilon) \Leftrightarrow \bot) \wedge (\zeta \Rightarrow \upsilon^{\swarrow}) \wedge (\xi^{\nearrow\swarrow} \Leftrightarrow \zeta^{\nearrow\swarrow}) \wedge (\xi \Rightarrow (\zeta^{\nearrow} \wedge (\upsilon^{=\swarrow})^{\swarrow}) \backslash \zeta)}{Error^{\upsilon}\,\zeta \xrightarrow{\langle\rangle} Error^{\upsilon}\,\xi}$$

$$\textsc{Tnerr}\ \frac{((\zeta \wedge \upsilon^{\swarrow}) \Leftrightarrow \bot) \wedge (\xi^{\nearrow\swarrow} \Leftrightarrow \zeta^{\nearrow\swarrow}) \wedge ((\xi \Leftrightarrow \zeta^{ceil}) \vee (\xi \Rightarrow ((\zeta^{\nearrow} \backslash \zeta) \backslash \zeta^{ceil})))}{Error^{\upsilon}\,\zeta \xrightarrow{\langle\rangle} Error^{\upsilon}\,\xi}$$

$$\textsc{Tres}\ \frac{t\,\zeta \xrightarrow{\mathcal{S}} t'\,\zeta'}{(\nu a\ t)\,\zeta \xrightarrow{\mathcal{S}\backslash a} (\nu a\ t')\zeta'}$$

$$\textsc{Tsum}\ \frac{t\,\zeta^{\downarrow x} \xrightarrow{\mathcal{S}} t'\,\zeta' \quad u\,\xi^{\downarrow x} \xrightarrow{\mathcal{S}'} u'\,\xi'}{(t+u)\,(\zeta \wedge \xi) \xrightarrow{\mathcal{S},\mathcal{S}'} (t'+u')\,(\zeta' \wedge \xi')\backslash x}\ \Delta$$

$$\textsc{Tpar}\ \frac{t\,\zeta^{\downarrow x} \xrightarrow{\mathcal{S}} t'\,\zeta' \quad u\,\xi^{\downarrow x} \xrightarrow{\mathcal{S}'} u'\,\xi'}{(t \mid u)\,(\zeta \wedge \xi) \xrightarrow{\mathcal{S},\mathcal{S}'} (t' \mid u')\,(\zeta' \wedge \xi')\backslash x}\ \nabla$$

$C \triangleq (clocks(\zeta) \cap clocks(\xi)) \cup \{x\}$
$\Delta \triangleq ((\zeta \wedge \xi) \not\Leftrightarrow \bot) \wedge ((\zeta' \wedge \xi') \not\Leftrightarrow \bot) \wedge x \notin (clocks(\zeta) \cup clocks(\xi)) \wedge (\zeta'^{/C} \Leftrightarrow \xi'^{/C})$.
$\nabla \triangleq \Delta \wedge ((\mathcal{S} = \langle \mathcal{R} : \mathcal{O}\rangle \wedge \mathcal{S}' = \langle \mathcal{R}' : \mathcal{O}'\rangle) \Rightarrow (\overline{\mathcal{R}} \cap \mathcal{O}' = \emptyset \wedge \overline{\mathcal{R}}' \cap \mathcal{O}' = \emptyset \wedge \overline{\mathcal{R}} \cap \mathcal{R}' = \emptyset))$

Table 4.4: Symbolic Transitional Semantics for π^δ: part 3

of downward most constraining clocks $\mathcal{C}^{\downarrow}$ if and only if $\zeta_\wedge \Rightarrow x > p$ for some p and, for any clock $y \neq x$ and constant q, if $\zeta_\wedge \Rightarrow y > q$, then $\zeta_\wedge \Rightarrow x - y \leq p - q$. The *upward border* $\zeta_\wedge^{\uparrow=}$ of $\zeta_\wedge$ is the least conjunctive constaint satisfying, for any clocks x and y and any constant p:

- $\zeta_\wedge^{\uparrow=} \Rightarrow (x = p)$ if $x \in \mathcal{C}^{\uparrow} \wedge \zeta_\wedge \Rightarrow x < p$,
- $\zeta_\wedge^{\uparrow=} \Rightarrow (x \sharp p)$ if $x \notin \mathcal{C}^{\uparrow} \wedge \zeta_\wedge \Rightarrow x \sharp p$, and
- $\zeta_\wedge^{\uparrow=} \Rightarrow x - y \sharp p$ if $\zeta_\wedge \Rightarrow x - y \sharp p$.

The *downward border* $\zeta^{=\downarrow}$ of a region constraint $\zeta_\wedge$ is defined in a symmetric fashion, merely replacing $\mathcal{C}^{\uparrow}$ by $\mathcal{C}^{\downarrow}$ and "$<$" by "$>$" in the previous definition. Those definitions are extended to some general constraint ζ in the usual way: if $\zeta \triangleq \zeta_{\wedge 1} \vee \cdots \vee \zeta_{\wedge k}$ for some k, then $\zeta^{\uparrow=} \triangleq \zeta_{\wedge 1}^{\uparrow=} \vee \cdots \vee \zeta_{\wedge k}^{\uparrow=}$, and $\zeta^{=\downarrow} \triangleq \zeta_{\wedge 1}^{=\downarrow} \vee \cdots \vee \zeta_{\wedge k}^{=\downarrow}$.

The *ceiling* ζ^{ceil} of a zone ζ defined as the latest unbounded zone that is reachable from ζ by letting time pass. It is obtained as the fixpoint $\zeta = F^t(\zeta)$ of an auxilliary function F, first defined on strictly conjunctive constraints $\zeta_\wedge$ as the weakest constraint verifying the following predicate: if there is a clock $x \in clocks(\zeta_\wedge)$ such that for all $y \in clocks(\zeta_\wedge)$, $(\zeta_\wedge^{\uparrow} \wedge x = ceil(x, \zeta)) \Rightarrow y \leq ceil(y, \zeta)$ then

- $F^t(\zeta_\wedge) \Rightarrow (x \geq ceil(x, \zeta))$,
- $F^t(\zeta_\wedge) \Rightarrow (x \sharp p)$ if $\zeta_\wedge \Rightarrow x \sharp p$, and
- $F^t(\zeta_\wedge) \Rightarrow x - y \sharp p$ if $\zeta_\wedge \Rightarrow x - y \sharp p$;

the definition is extended to a general constraint $\zeta \triangleq \zeta_{\wedge 1} \vee \cdots \vee \zeta_{\wedge k}$ by $F^t(\zeta) \triangleq F^t(\zeta_{\wedge 1}) \vee \cdots \vee F^t(\zeta_{\wedge k})$. The existence of the fixpoint can be proven by taking for measure the number of clocks which are not forced to be either equal or strictly superior to ceil(t) in any $\zeta_{\wedge i}$, $i \in \{1..k\}$.

Having gathered the needed paraphernalia, we can now precisely describe the fundamental point supporting each rule of Table 4.3. In TSEL, it is required that no concrete process in zone $([\sigma, \upsilon]\pi.\, t)\, \zeta$ may perform action π (because $(\zeta \wedge \sigma^{\uparrow}) \Leftrightarrow \bot$), but that all will acquire this capability by letting time pass: *all* the concrete processes in the zone (because $\zeta^{\uparrow\downarrow} \Rightarrow \sigma^{\uparrow\downarrow}$) will eventually be able to perform π if the values of the clocks constrained by ζ grow sufficiently (because the two preceding conditions imply that $(\sigma \wedge \zeta^{\uparrow}) \not\Leftrightarrow \bot$). The region that is reached is then precisely the lower border of $(\sigma \wedge \zeta^{\uparrow})$, that to say the proximate frontier of time discontinuity.

The rule TURG is similar to TSEL, but it allows time progression from a zone where the action prefix $[\sigma, \upsilon]\pi.\, t$ is *selected* (but *not urgent* due to the condition $\zeta \Rightarrow (\sigma \backslash \upsilon^{\uparrow})$) to a zone that is the discontinuity frontier where that prefix *becomes* urgent. The rule TELA allows time to progress from a zone where an action is selected to a zone where it is not, at the condition that the selection region is not unbounded ($\sigma^{\uparrow} \not\Leftrightarrow \sigma$) and the action may not become urgent at any time in the future ($(\upsilon \wedge \zeta^{\uparrow}) \Leftrightarrow \bot$). The region that can be reached is the discontinuity frontier between the zones where π is enabled, and the posterior zones where it is not.

The rule TIDL is the natural follow-up of TELA, allowing to reach a zone where time progress is unbounded from a zone where the action π is not selected except

(possibly) for its downward border, if π may not be selected at any point in the future. Then, the ready set carried by the transition is logically empty. The rule TUNBS treats the alternative case, that is when the selection condition σ is unbounded ($\sigma^{\uparrow} \Leftrightarrow \sigma$). The condition σ will then never become false in the future, and unbounded progress can be accomplished without crossing any discontinuity frontier. Similar is TUNBU, in a zone where the urgency condition is unbounded.

The rules accurately describe how time may progress in our models, being even *fully abstract* in a sense. They indeed give in any situation the *maximal* progression a process can make in one time-passing step. This would be correct if we had to consider the actions prefixes of a term separately, without making any correlations among them. But the action prefixes put in parallel or in the alternatives of a choice within a given term *do have* an influence on each other's time progression. The situation can be represented as if they were playing the following fictitious game: each prefix announces the amount of time it agrees to let go, the prefix announcing the least amount of time winning the game, meaning that its forces all the other prefixes to respect its pace. Hence when an abstract process $t\,\zeta$ agrees to let time elapse up to some zone $t'\,\zeta'$, it should actually be ready to limitate its progression to any zone between $t\,\zeta$ and $t'\zeta'$, in case another prefix would win the game we just described. This is a usual way of defining real-time semantics in process algebras (see for example [NS92] and [LL98]). The game-theoretic formulation was inspired by [GSSAL94].

This is what rule TREG does. It applies only to transitions where the term of the destination zone is not $Error$, the similar treatment for urgent action being achieved through rule TREQ. So, if $t\,\zeta \xrightarrow{\mathcal{S}} t'\,\zeta'$, then TREG allows to infer transitions to regions ξ comprised in the time frame strictly between ζ and ζ' (with ζ' and the lower border of ζ excluded), which is accurately written $(\xi \Rightarrow ((\zeta^{\uparrow} \backslash \zeta^{=\swarrow})\zeta'^{\uparrow}))$. An additional condition imposes that ξ has the same width as ζ, so that all the concrete processes of $t\,\zeta$ can perform the transition to $t\,\xi$.

The rule TMISS precises the conditions under which a process may produce an error. For that, its urgency condition must have an upper bound, through which it limitates time progression. The rule TREQ plays the same role as rule TREG, but in the case where the action remains urgent during the transition.

The rule TERR deals with explicit error processes. It may apply in the case where all the valuations in the current zone may validate the condition υ by letting time pass. Any zone between the current one and the lower border of υ is then reachable.

Finally, the rules TSUM and TPAR require that both processes in a choice or a composition agree on the time that may pass, as illustrated by the fictitious game given above. The mechanism ensuring this, involves a fictitious clock that is set to zero in the constraints of the processes that are used in the antecedents of the rules. We require by a side condition on both rules that this clock is new to constraints ζ and ξ, while the reachable constraints ζ' and ξ' agree exactly on the value reached by x: $\zeta'^{/x} \Leftrightarrow \xi'^{/x}$. This method has probably been inspired to us by [BJLY98].

If one would try to embrace the general structure of the abstract transition system we just described, he or she would surely notice that certain regions have time-passing transitions that leave from them and return to them (the so-called *self-loops*). Specifically, on any region which does not constrains one or more clocks

to equal some constant value. Those loops are obtained through rules TREG and TREQ. Those self-loops are necessary in our semantics to properly deal with choice and parallel compostition: they ensure the completeness of rules TSUM and TPAR (see the proof of Theorem 4.2.2). They are also the sign that time-abstracting bisimulation does not preserve non-zenoness [TY01]. This drawback is however not a hindrance to the decidability of timed late bisimilarity, because if one sufficiently refines the state space, processes with zeno and without zeno executions can again be distinguished.

In conclusion to this subsection, we also notice that each abstract transition represents a potentially infinite number of concrete transitions. Hence, our abstract TTSs are not only image finite, but also *finitely branching*. This allows us to provide the sound and complete proof system that will follow.

4.1.5 Semantics of a Small Example (Continued)

We consider again the example given in Section 3.7:

$$t \equiv \nu a\ (\nu b\ \overline{a}b.\, \nu y\ [y \leq 6, y < 6]b() \mid a(c).\, \nu x\ [x \geq 5, \bot]\overline{c}())\ .$$

We will show how some abstract transitions can be inferred from the rules on process $t\top$. As in the concrete case indeed, the initial value of any clock that could be considered for the system is irrelevant. We also note $u \equiv \nu y\ [y \leq 6, y < 6]b()$ and $v \equiv \nu x\ [x \geq 5, \bot]\overline{c}()$.

Thus, initially $t\top$ may only time pass at will, the axiom that applies then is TUNB, the *true* selection condition $\top$ being unbounded: $(\top^{\nearrow} \Leftrightarrow \top)$. Hence we can write:

$$\dfrac{\dfrac{\dfrac{\dfrac{(\top \Rightarrow \top)\ \wedge\ (\top^{\nearrow} \Leftrightarrow \top)}{(\overline{a}b.\, u)\top \xrightarrow{\langle \emptyset : \overline{a}^p \rangle} (\overline{a}b.\, u)\top}\ \textsc{Tunb}}{(\nu b\ \overline{a}b.\, u)\top \xrightarrow{\langle \emptyset : \overline{a}^p \rangle} (\nu b\ \overline{a}b.\, u)\top}\ \textsc{Tres} \qquad \dfrac{\dfrac{(\top \Rightarrow \top)\ \wedge\ (\top^{\nearrow} \Leftrightarrow \top)}{(a(c).\, v)\top \xrightarrow{\langle \emptyset : a^p \rangle} (a(c).\, v)\top}\ \textsc{Tunb}}{\dfrac{(\nu b\ \overline{a}b.\, u \mid a(c).\, v)\top \xrightarrow{\langle \emptyset : a^p, \overline{a}^p \rangle} (\nu b\ \overline{a}b.\, u \mid a(c).\, v)\top}{t\top \xrightarrow{\langle\rangle} t\top}\ \textsc{Tres}}\ \textsc{Tpar}$$

Of course, the reader is invited to compare the above proof tree with the one in Section 3.7; they are extremely alike. An interaction on port a may now happen if it respects the constraints on name matching; we only show by this simple case that the rules of Table 4.2 work perfectly as in the concrete case when processes have no

free names. We suppose again $u' \equiv \nu y\ [y \leq 6, y < 6]c()$, we can write:

$$\textsc{Close}\ \dfrac{\textsc{Open}\ \dfrac{\textsc{Conv}\ \dfrac{\textsc{Pre}\ \dfrac{}{\top \Rightarrow \top}}{(\overline{a}c.\, u')\top \xrightarrow{\overline{a}c,\top} u'\top}}{(\nu c\ \overline{a}c.\, u')\top \xrightarrow{\overline{a}(c),\top} u'\top} \qquad \textsc{Pre}\ \dfrac{}{\top \Rightarrow \top}}{(\nu b\ \overline{a}b.\, u)\top \xrightarrow{\overline{a}(c),\top} u'\top \qquad (a(c).\, v)\top \xrightarrow{a(c),\top} v\top}$$

$$\textsc{Res}\ \dfrac{(\nu b\ \overline{a}b.\, u \mid a(c).\, v)\top \xrightarrow{\tau,\top} (\nu c\ (u' \mid v))\top}{t\top \xrightarrow{\tau,\top} \nu a\ (\nu c\ (u' \mid v))\top}$$

After the initial interaction on a, the time progression for the process which term encloses sub-terms u' and v put in parallel is bounded by the time progression possible for the subterm v; assuming that $\zeta_{xy} \Leftrightarrow (x = 5 \wedge y = 5 \wedge x = y)$ and similarly $\zeta_{xz} \Leftrightarrow \zeta_{xy}[z/y]$, $\zeta_{yz} \Leftrightarrow \zeta_{xy}[z/x]$, we can infer from Tpar:

$$\textsc{Tres}\ \dfrac{\textsc{Tpar}\ \dfrac{u'\top^{\downarrow z} \xrightarrow{\langle c:\emptyset\rangle} u'\,\zeta_{yz} \quad v\top^{\downarrow z} \xrightarrow{\langle\rangle} v\,\zeta_{xz}}{(u' \mid v)\top \xrightarrow{\langle c:\emptyset\rangle} (u' \mid v)\,\zeta_{xy}}}{(\nu a\ (u' \mid v))\top \xrightarrow{\langle\rangle} (\nu a\ (u' \mid v))\,\zeta_{xy}}$$

Then, using the rule Tsel, the needed transition on v can be inferred by using the following equivalences: $\top^{\downarrow z\downarrow x} \Leftrightarrow (z = 0 \wedge x = 0)$, $(x \geq 5)^{\uparrow} \Leftrightarrow (x \geq 5)$, $(z = 0 \wedge x = 0)^{\uparrow} \Leftrightarrow (z \geq 0 \wedge x \geq 0 \wedge (x = z))$, $(z = 0 \wedge x = 0)^{\uparrow\downarrow} \Leftrightarrow x = z$, and $(x \geq 5)^{\uparrow\downarrow} \Leftrightarrow \top$.

$$\textsc{Reset}\ \dfrac{\textsc{Tsel}\ \dfrac{((\top^{\downarrow z\downarrow x} \wedge x \geq 5) \Leftrightarrow \bot) \wedge ((\top^{\downarrow z\downarrow x})^{\uparrow\downarrow} \Rightarrow \top) \wedge ((x \geq 5 \wedge (\top^{\downarrow z\downarrow x})^{\uparrow}) \not\Leftrightarrow \bot)}{([x \geq 5, \bot]\overline{c}())\top^{\downarrow z\downarrow x} \xrightarrow{\langle\rangle} ([x \geq 5, \bot]\overline{c}())\ ((\top^{\downarrow z\downarrow x})^{\uparrow} \wedge (x \geq 5))^{=\downarrow}}}{v\top^{\downarrow z} \xrightarrow{\langle\rangle} v\,\zeta_{xz}}$$

Producing the counterpart deduction on subterm u' involves the rule Treq, while noticing that $((\top^{\downarrow z\downarrow y})^{\uparrow} \setminus \top^{\downarrow z\downarrow x}) \Leftrightarrow (z > 0 \wedge y > 0)$, $(\top^{\downarrow z\downarrow y})^{\uparrow\downarrow} \Leftrightarrow (y = z)$, and $\zeta_{yz}^{\uparrow\downarrow} \Leftrightarrow (y = z)$:

$$\textsc{Reset}\ \dfrac{\textsc{Treq}\ \dfrac{(\top^{\downarrow z\downarrow y} \Rightarrow y < 6) \wedge (\zeta_{yz} \Rightarrow (y < 6 \wedge (z > 0 \wedge y > 0))) \wedge ((\top^{\downarrow z\downarrow y})^{\uparrow\downarrow} \Leftrightarrow \zeta_{yz}^{\uparrow\downarrow})}{([y \leq 6, y < 6]c())\top^{\downarrow z\downarrow y} \xrightarrow{\langle c:\emptyset\rangle} ([y \leq 6, y < 6]c())\,\zeta_{yz}}}{u'\top^{\downarrow z} \xrightarrow{\delta\langle c:\emptyset\rangle} u'\,\zeta_{yz}}$$

Using the rule Treg on v and other applications of the rule Treq on u', one may also prove that all the regions which are comprised between $(x > 0 \wedge y > 0)$ and $\zeta_{xy} \Leftrightarrow (x < 5 \wedge y < 5 \wedge x = y)$ are reachable.

To the question: "*can this process lead to an error*?" the answer is (as wanted) negative consistently with the concrete case. There again, it is easy to see that, after the discontinuity frontier ζ_{xy} has been reached, no time-passing transition is fireable, because of the side condition of rule Tpar forbidding ready sets with common names in one another's requirement/offer sets.

4.2 Algebraic Laws for Processes

Having finally settled all the semantic issue that uprose along the path, we may now care to actually provide some results on the axiomatization of timed late bisimulation (of Definition 4.1.1) for processes with finite (*i.e.* recursion-impaired) terms. The developments are organized in the following way. We start by exploring the properties that link our abstract and concrete transition systems. This exploration achieved, we propose a symbolic timed late bisimulation relation on abstract processes. We show that proving symbolic timed late bisimulation on abstract processes is equivalent to proving timed late bisimulation for a potentially infinite set of concrete processes. Finally, we propose a proof system for symbolic timed late bisimulation on processes with finite terms, providing soundess and completeness proofs.

4.2.1 Relating Concrete and Abstract Transition Systems

There are two essential theorems that we show in this section: to an abstract transition correspond an infinite number of concrete transitions (*soundness* of the abstraction), and for a concrete transition there exists a zone in which the abstract context can perform a corresponding transition (a form of *weak completeness* of the abstraction). Those results are fundamental to prove the soundness and completeness of the proof system in Section 4.2.3, and also for the behavioral type system of the next chapter.

Theorem 4.2.1 (Soundness of the Abstraction).
For any $t, t', \zeta, \zeta', \mathcal{S}, \pi, \mu$, we have:

$$t\,\zeta \xrightarrow{\mathcal{S}} t'\,\zeta' \Rightarrow (\forall \rho, \delta.\ (\rho \vDash \zeta \wedge \rho^{+\delta} \vDash ((\zeta^{\nearrow} \backslash \zeta^{=\swarrow}) \wedge \zeta'^{\swarrow})) \Rightarrow t\rho \xrightarrow{\delta\,\mathcal{S}} t'\rho^{+\delta})$$

and

$$\begin{aligned} t\,\zeta \xrightarrow{\pi,\mu} t'\,\zeta' \Rightarrow ((\forall \rho, \alpha.\ ((bn(\pi) \cap (fn(t) \cup ports(\alpha))) = \emptyset \wedge \rho \vDash \zeta \wedge \alpha \vDash \mu) \\ \Rightarrow (\exists \rho'.\ \rho' \vDash \zeta' \wedge (t\alpha)\rho \xrightarrow{\pi\alpha} (t'\alpha)\rho')) \wedge \\ (\forall \rho', \alpha.\ ((bn(\pi) \cap (fn(t) \cup ports(\alpha))) = \emptyset \wedge \rho' \vDash \zeta' \wedge \alpha \vDash \mu) \\ \Rightarrow (\exists \rho.\ \rho \vDash \zeta \wedge (t\alpha)\rho \xrightarrow{\pi\alpha} (t'\alpha)\rho'))) \ . \end{aligned}$$

Proof of Soundness. By routine induction on the structure of the operational semantics deduction trees. The proof of the timed case is similar to the first clause of [LY02, Lemma 4]. The proof for the discrete case is an adaptation of the proof for the second clause of [Lin94, Lemma 2.12]. □

Theorem 4.2.2 (Weak Completeness of the Abstraction).
For any $t, t', \rho, \rho', \mathcal{S}, \pi, \delta, \alpha$, we have:

$$t\rho \xrightarrow{\delta\,\mathcal{S}} t'\rho' \Rightarrow (\exists\, \zeta.\ \rho \vDash \zeta \wedge (\rho^{+\delta} \vDash \zeta \vee (\exists\, \zeta'.\ \rho^{+\delta} \vDash \zeta' \wedge t\,\zeta \xrightarrow{\mathcal{S}} t'\,\zeta')))$$

and

$$\begin{aligned} &((t\alpha)\rho \xrightarrow{\pi\alpha} t'\rho' \wedge (bn(\pi\alpha) \cap fn(t\alpha)) = \emptyset \wedge ports(\mu) \subseteq (fn(t) \cup fn(u))) \\ &\Rightarrow (\exists \zeta, \zeta', \mu, t''.\ (\rho \vDash \zeta \wedge \rho' \vDash \zeta' \wedge \alpha \vDash \mu \wedge t' \equiv t''\alpha \wedge t\,\zeta \xrightarrow{\pi,\mu} t''\,\zeta')) \ . \end{aligned}$$

Proof of Weak Completeness. The proof of the first clause is similar to the proof for the second clause of [LY02, Lemma 4]. The proof for the discrete case is similar to the first clause of [Lin94, Lemma 2.12]. □

We shall actually need those results in the next chapter. For *finite* terms (which are targeted by our proof system), we conjecture that it can be shown that the abstract transition system corresponds to a safe *abstract interpretation* of the concrete transition system. In such a case, processes are organized as a cpo which order is given by reduction, and bisimulation is defined (co-)inductively as the least fixed point of a monotonous function over processes. The time-abstracting ground bisimulation provides an abstraction that, together with a concretion function associating a concrete process to each abstract process, forms a *galois connexion*. This allows to relate the abstract and concrete domains, as the previous theorems do. Yet, results given by abstract interpretation then apply, and the existence of a fix-point for a monotonous function at each level leads to the existence of a fix-point at the other level and an approximation of its value. In our case, this means that for any model, to find a symbolic timed late bisimulation (defined in the next section) implies the existence of a (concrete) timed late bisimulation. Well-know results on the abstract interpretation of timed systems can be found in [HPR97, DH95].

4.2.2 Symbolic Timed Late Bisimulation

The fact that we are unable to prove stronger theorems than Theorems 4.2.1 and 4.2.2 actually suggests an unfortunate (though expected) deficiency of our abstract semantics. Namely, for any zone $t\,\zeta$, it is *false* to affirm that $(\rho \vDash \zeta \wedge \kappa \vDash \zeta) \Rightarrow (t\rho \sim_{tag} t\kappa)$. This result would have allowed us to reduce finding a timed late bisimulation on concrete processes to finding a late bisimulation on time-abstracted processes, as shown for example in [TY01]. Yet, this does not mean such a result can not be obtained: it only means that finding a late bisimulation on abstract processes is not sufficient to guarantee timed late bisimulation on concrete processes; we need a more powerful (*i.e.* distinctive) notion of equivalence on abstract processes.

In a nutshell, the wanted result fails because some zones may be too coarse; we therefore need a notion of bisimulation on abstract processes that is allowed to refine the (abstract) state space at will.

We typically want to know if for some ζ and ξ, we have for all ρ and κ the property $(\rho \vDash \zeta \wedge \kappa \vDash \xi) \Rightarrow t\rho \sim_{tl} u\kappa$. Consider the usual view of bisimulation as the copycat game between two players. If the protagonist $t\,\zeta$ may perform $\xrightarrow{\pi,\mu}$ (*i.e.* $t\rho \xrightarrow{\pi,\mu} t'\rho'$ for any $\rho \vDash \zeta$), we have to determine if the antagonist $u\,\xi$ is able to answer at any moment $\kappa \vDash \xi$ with a "similar" or "compatible" transition. Hence, if all $\kappa \vDash \xi$ are not time-abstracting bisimilar, ξ needs to be cut into pieces: $u\,\xi$ is not able to match the discrete transition $\xrightarrow{\pi,\mu}$ of $t\,\zeta$, but there may be a partition $\xi_1, \xi_2, \cdots, \xi_k$ of ξ so that each zone $u\,\xi_i$ can perform $\xrightarrow{\pi,\mu}$. As $\xi_1, \xi_2, \cdots, \xi_k$ is a partition of ξ, we have $(\bigvee_{i\in\{1..k\}} \xi_i) \Leftrightarrow \zeta$, hence proving that $u\,\xi$ has a matching transition for all $\rho \vDash \xi$.

This is fine, but the above description ignores naming issues. They are dealt with by applying the exact technique devised by Lin in [Lin94]. This technique allows, when a process receives a port name from its environment, to predict its behavior

by studying only a finite number of transitions, although the set of names that can be received is infinite. The main consequence is that each transition then has a different name matching set: if $t\,\zeta \xrightarrow{\pi,\eta} t'\zeta'$, then for all the maximal matching sets θ consistent with η, $u\,\xi$ must have a transition $\xrightarrow{\pi',\theta'}$ such that $\theta \Rightarrow \theta'$ and $\pi \equiv^{\theta} \pi'$. Since for any matching set there are only finitely-many maximally consitent sets over a given finite set of variables V, we only have to examine a finite number of matching sets by taking $V = \mathrm{fn}(t) \cup \mathrm{fn}(u)$.

The following definition is therefore close enough to that of Lin [Lin94], but it differs from the definitions of symbolic timed bisimulations as seen in [LY00] and [BD94]. The distinguishing element is the presence of abtracted time-passing actions in our model. To accurately judge that processes let the same amount of time elapsed, they are forced to synchronize on a fictitious new clock, as we did for abstract parallel composition. To do so, we have to give two ancillary definitions, starting with the *slice partition* of a zone.

Definition 4.2.1 (Slice Partition). *A set of indexed zone constraints $\zeta_1, \cdots, \zeta_k$ is a slice partition for a zone $t\,\zeta$ if and only if $(\zeta_1 \wedge \cdots \wedge \zeta_k) \Leftrightarrow \bot$, $(\zeta_1 \vee \cdots \vee \zeta_k) \Leftrightarrow \zeta$, for all $1 \leq i \leq k$ we have $\zeta_i^{\uparrow\downarrow} \Leftrightarrow \zeta^{\uparrow\downarrow}$, and whatever $\zeta_i' \Rightarrow \zeta_i$ then $t\,\zeta_i' \xrightarrow{\mathcal{S}} t'\,\zeta_i''$ implies $t\,\zeta_i \xrightarrow{\mathcal{S}} t'\,\zeta_i'''$ with $\zeta_i'' \Rightarrow \zeta_i'''$.*

Definition 4.2.2 (Zone of Constant Width). *A zone ζ is of constant width p if and only if whenever*

$$((\bigwedge_{y,y' \in clocks(\zeta)} (y - y' = q_{y,y'})) \wedge \zeta) \not\Leftrightarrow \bot$$

for some clock-indexed set of values ranged over by $q_{y,y'}$, we have for any $z \in clocks(\zeta)$ that

$$((\bigwedge_{y,y' \in clocks(\zeta)} (y - y' = q_{y,y'})) \wedge \zeta)^{/z} \Leftrightarrow (q \preceq z \preceq q')$$

implies $q' - q = p$.

Definition 4.2.3 (Symbolic Timed Late Bisimulation). *A matching-set indexed family of symmetric relation R^{μ} between processes $t\,\zeta$ and $u\,\xi$ with $ports(\mu) \subseteq (fn(t) \cup fn(u))$ is a symbolic timed late bisimulation if and only if for any $x \notin clocks(t) \cup clocks(u)$, $(t\,\zeta, u\,\xi) \in R^{\mu}$ implies that for any slice partition $\zeta_1, \cdots, \zeta_k$ of $(\zeta^{\downarrow x\uparrow} \wedge \zeta)$, there exists a slice partition $\xi_1, \cdots, \xi_k$ of $(\xi^{\downarrow x\uparrow} \wedge \xi)$ such that each ζ_i and ξ_i have the same constant width, $(t\,\zeta_i^{\backslash x}, u\,\xi_i^{\backslash x}) \in R^{\mu}$, and:*

- *if $t\,\zeta_i \xrightarrow{\mathcal{S}} t'\zeta'$ then $u\,\xi_i \xrightarrow{\mathcal{S}'} u'\xi'$ with $\mathcal{S} \preceq \mathcal{S}'$, $t'\,\zeta'$ and $u'\,\xi'$ have the same constant width, $\zeta_i^{/x} \Leftrightarrow \xi_i^{/x}$, and $(t'\,\zeta'^{\backslash x},\ u'\,\xi'^{\backslash x}) \in R^{\mu}$;*
- *there exists a $(\zeta_i^{\backslash x} \wedge \xi_i^{\backslash x})$-partition $\zeta\xi_1, \cdots, \zeta\xi_l$ ranged over by $\zeta\xi_j$, such that whenever $t\,(\zeta\xi_j)^{/clocks(\zeta)} \xrightarrow{\pi,\eta} t''\,\zeta''$ with $(bn(\pi) \cap (fn(t) \cup fn(u))) = \emptyset$, then for*

each $\theta \in MCE_{fn(t)\cup fn(u)}(\mu \wedge \eta)$ *there is a transition* $u\,(\xi_j)^{/clocks(\xi)} \xrightarrow{\pi',\theta'} u'\,\xi'$ *so that* $\theta \Rightarrow \theta'$, $\pi \equiv^{\theta} \pi'$, *and* $(t''\,\zeta'',\ u\,\xi') \in R^{\theta''}$, *where*

$$\theta'' \triangleq \begin{cases} \theta \wedge \bigwedge_{c \in (fn(\pi.t) \cup fn(\pi'.u))} b \neq c & \text{if } \pi = \overline{a}(b) \text{ for some } a \\ \theta & \text{otherwise.} \end{cases}$$

If, for some matching set μ, *there exists a relation* R^μ *between* $t\,\zeta$ *and* $u\,\xi$, *then we note* $t\,\zeta \sim^{\mu}_{stl} u\,\xi$. *We note* $t\,\zeta \sim_{stl} u\,\xi$ *if and only if there exists a matching set* μ *for which* $t\,\zeta \sim^{\mu}_{stl} u\,\xi$. *This relation hence forms the largest timed late bisimulation over processes; it is called* symbolic timed late bisimilarity. *If the relation is not symmetric, then it defines a preorder called* symbolic timed late simulation, *noted* $u\,\xi \preceq^{\mu}_{stl} t\,\zeta$. *The largest such simulation is called* symbolic timed late similarity *and noted* $u\,\xi \preceq_{stl} t\,\zeta$.

We now informally discuss two consequences of using this notion of equivalence as implementation relation for our processes. We can reason either at the abstract or the concrete level, since we just saw in Section 4.2.1 that they strongly correspond one to the other. First, if we consider only processes with prefixes of the form $[\top, \bot]\pi$, then our timed late bisimulation corresponds to the strong bisimulation for the π-calculus [MPW92], and our symbolic timed late bisimulation to the symbolic bisimulation of [Lin94]. Second, our bisimulation fails to be a congruence for processes. This was yet expected, since early and late bisimulations for the π-calculus are themselves not preserved by input prefix.

We provide the follow-up theorem to Theorems 4.2.1 and 4.2.2, relating symbolic timed late bisimulation and timed late bisimulation:

Theorem 4.2.3. *For all* t, u, ζ, ξ, μ: $t\,\zeta \sim^{\mu}_{stl} u\,\xi$ *if and only if, for any* $\alpha \vDash \mu$, $\rho \vDash (\zeta^{\downarrow x\uparrow} \wedge \zeta)$ *and* $\kappa \vDash (\xi^{\downarrow x\uparrow} \wedge \xi)$ *with* $x \notin (clocks(\zeta) \cup clocks(\xi))$ *and* $\rho(x) = \kappa(x)$, *we have* $(t\alpha)\rho^{\backslash x} \sim_{tl} (u\alpha)\kappa^{\backslash x}$.

Proof (of Theorem 4.2.3). The proof is a combination of the proofs of item 1 in [Lin94, Proposition 2.13] and of [LY02, Theorem 12]. The essential difference is that in our case we do not have the time continuity property, and that we must therefore cut the state space into continuous pieces to achieve the proof. □

4.2.3 A Proof System for Terms with Finite Control

We now propose a sound and complete proof system, which rules are given in Tables 4.5 and 4.6. Processes proved equivalent using those rules may not have terms featuring parallel compositon or recursion. Parallel composition can however be dealt with through an *expansion law* that allows the replacement of concurrency by non-determinism. Again, our proof system draws matter from the one provided by Lin in [Lin94] to prove late bisimilarity on π-calculus terms: Lin's technique is used to deal with naming issues, while time-related aspects are treated in an ad-hoc fashion.

The judgments of our proof system are of the form:

$$\mu : \phi, \psi \rhd t\,\zeta = u\,\xi\ ,$$

where ϕ and ψ are simple time constraints of the form

$$\psi ::= \phi ::= x \leq q \mid x < q \mid \top \mid \bot ,$$

x being a clock name, and q being a natural constant.

The meaning of those judgements is based on a notion of *timed bisimulation up-to a given delay* δ_0: two processes are timed bisimilar up-to δ_0 if all the processes reachable by letting a time $\delta \leq \delta_0$ elapse are bisimilar. Hence, all the processes of two zones $t\,\zeta$ and $u\,\zeta$ are timed bisimilar up-to δ_0 if

- they can perform the same discrete actions and the resulting processes are themselves timed bisimilar, and if
- whenever one of them can let a delay $\delta \leq \delta_0$ pass, so can the other, and the resulting processes are bisimilar up-to $\delta_0 - \delta$.

Timed bisimilarity is the largest such timed bisimulation. We trivially obtain that timed bisimilarity up-to infinity and timed bisimilarity simply coincide. Furthermore, under certain circumstances, timed bisimilarity up-to a finite delay δ_0 and up-to infinity also coincide. This is true at least in the case where the processes under scrutiny may not let time progress past δ_0. This property is essential because it will allow us to establish the completeness of our proof system.

But let us go back to the meaning of a judgment $\mu : \phi, \psi \rhd t\,\zeta = u\,\xi$. It states that, under the name matching constraint μ, $t\,\zeta$ and $u\,\xi$ are timed bisimilar up-to the highest delay maintaining the validity of ϕ, and that this judgment can be soundly used to prove timed bisimilarity up-to infinity by proving timed bisimilarity up-to ψ. If ϕ is true, proving the judgment conspicuously means proving timed bisimilarity. Otherwise, we will say that a judgement proves bisimilarity *up-to* ϕ. We shall write $\mu \rhd t\,\zeta = u\,\xi$ as a shortcut for $\mu : \top, \top \rhd t\,\zeta = u\,\xi$. At this point, the role of the other constraint ψ may still seem veiled by obscurity. We next alleviate this shortcoming by elaborating on a few words.

A judgment with a constraint ψ indicates that both processes $t\,\zeta$ and $u\,\xi$ either will not let time progress so that ψ becomes false, or will let ψ become false but then deadlock. Hence ψ is used in the Choice rule of Table 4.6: if $t\,\zeta$ and $u\,\xi$ prevent time from progressing past the point where ψ is true, then it is useless to check that processes $t'\,\zeta$ and $u'\,\xi$ are bisimilar up-to infinity so that $(t + t')\,\zeta$ and $(u + u')\,\xi$ be bisimilar up-to infinity; it suffices to check that $t'\,\zeta$ and $u'\,\xi$ are bisimilar up-to ψ. Using $\mu : \top, \top \rhd t\,\zeta = u\,\xi$ as an antecedent in the Choice rule therefore requires comparing $t'\,\zeta$ and $u'\,\xi$ up-to infinity, whereas using $\mu : \top, \bot \rhd t\,\zeta = u\,\xi$ requires comparing $t'\,\zeta$ and $u'\,\xi$ only on the basis of their immediately triggerable discrete transitions.

We need such reasonings, employing ψ, to ensure the completeness of our proof system. On the one hand, ψ may only be used soundly on processes that prevent time from diverging; ψ is useless on other processes, because no axiom may then give it a value other than $\top$. On the other hand, lacking the capability of using ψ would make proving the equivalence of some processes impossible: among the processes that eventually block time, there are conspicuously many that have equivalent behaviors

when their clock context satisfies ψ but inequivalent behaviors when ψ has turned invalid.

A process may block time in two circumstances: either it has a silent transition with an urgency condition ending up when $x < q$ for some x and q, or it produces an error when $x \geq q$ for some x and q. Sound derivation trees using this property can therefore be built by employing rules Tau or Error of Table 4.6 and, afterwards, the Choice rule.

Just before we introduce the proof rules in a more formal and detailed fashion, we open a parenthesis to give some axioms that can be used as rewriting rules to reorganize terms by moving around port name restrictions and clock resets. Those axioms will be used to establish the completeness of the proof system by helping to shape-up process terms into some kind of *normal form.*

$$
\begin{array}{ll}
A1 & (t + t)\,\zeta = t\,\zeta \\
A2 & (\nu a\ [\sigma, \upsilon]\pi.\ t)\,\zeta = ([\sigma, \upsilon]\pi.\ \nu a\ t)\,\zeta \ \textbf{if}\ a \notin n(\pi) \\
A3 & (\nu x\ \nu y\ t)\,\zeta = (\nu x\ t[x/y])\,\zeta \\
A4 & (\nu m\ (t + u))\,\zeta = (\nu m\ t + \nu m\ u)\,\zeta \\
A5 & (\nu m\ \nu n\ t)\,\zeta = (\nu n\ \nu m\ t)\,\zeta \\
A6 & (\nu a\ [\sigma, \upsilon]\pi.\ t)\,\zeta = 0\,\zeta \ \textbf{if}\ (subj(\pi) = a) \wedge (\upsilon^\uparrow \Leftrightarrow \upsilon) \\
A7 & (\nu a\ [\sigma, \upsilon]\pi.\ t)\,\zeta = (Error^{(\upsilon'^{=})'})\,\zeta \ \textbf{if}\ (subj(\pi) = a) \wedge (\upsilon^\uparrow \not\Leftrightarrow \upsilon)
\end{array}
$$

The parenthesis closed, we now define the equality relation on processes as the least equivalence relation closed under the axioms and rules of Tables 4.5 and 4.6.

$$
\text{Equiv}\ \frac{t\,\zeta \equiv u\,\xi}{\mu \rhd t\,\zeta = u\,\xi} \qquad \text{Axiom}\ \frac{-}{\mu \rhd\ t\,\zeta = u\,\xi}\ \text{using}\ A1 - A7
$$

$$
\text{Res}\ \frac{\mu \wedge \left(\bigwedge_{b \in (\mathrm{fc}(t) \cup \mathrm{fc}(u))} a \neq b \right) : \phi, \psi \rhd t\,\zeta = u\,\xi}{\mu : \phi, \psi \rhd (\nu a\ t)\,\zeta = (\nu a\ u)\,\xi}\ a \notin ports(\mu)
$$

$$
\text{Match}\ \frac{\mu \wedge (a = b) : \phi, \psi \rhd t\,\zeta = u\,\xi \quad \mu \wedge (a \neq b) : \phi, \psi \rhd 0\,\zeta = u\,\xi}{\mu : \phi, \psi \rhd ([a = b]t)\,\zeta = u\,\xi}
$$

$$
\text{Part-}\mu\ \frac{\mu \wedge (a = b) : \phi, \psi \rhd t\,\zeta = u\,\xi \quad \mu \wedge (a \neq b) : \phi, \psi \rhd t\,\zeta = u\,\xi}{\mu : \phi, \psi \rhd t\,\zeta = u\,\xi}
$$

$$
\text{Conseq-}\mu\ \frac{\mu : \phi, \psi \rhd t\,\zeta = u\,\xi}{\mu' : \phi, \psi \rhd t\,\zeta = u\,\xi}\ \mu' \Rightarrow \mu \qquad \text{Absurd-}\mu\ \frac{-}{\bot : \phi, \psi \rhd t\,\zeta = u\,\xi}
$$

Table 4.5: Symbolic Proof System for π^δ: Part 1

The rules of Table 4.5 handle operations where updating name matching sets is

necessary. As a natural consequence, they appear to be independent of ϕ and ψ, their values being identical in the antecedent and the consequent of the rules. The only exceptions to that are of course the two axioms Equiv and Axiom, that identify congruent processes and processes that can be rewritten to the same form using the axioms $A1$ to $A7$ above. The identified processes are then bisimilar up-to infinity and we have to suppose that they do not prevent time from diverging, leading to $\phi = \top$ and $\psi = \top$.

Each rule Res, Match, Part-μ, Conseq-μ and Absurd-μ correspond to a rule in [Lin94]. The restriction rule Res requires the supposition that the introduced port name is free in processes on both sides of the equality. A matching operation can be solved either by identifying the compared names or by supposing them different. In the latter case, the failing comparison makes the left-most process behave as the idle one. The rule Part-μ is a sort of *excluded middle* allowing the partition of the name matching set in two sets, according to the agreement or disagreement on the equality of two names. Conseq-μ allows the strengthening of μ in the consequent. Absurd-μ implies that all processes are equivalent in antilogic environments.

The rules P-Input, C-Input, Output and Tau of Table 4.6 make use of predicate Equ_R on a set of rules R. Equ_R, employed a rule of our proof system such as Output, is true if and only if the zones from both parts of the inferred equality allow only compatible discrete transitions, and compatible time-passing transitions. The definition of Equ uses a predicate $const\_width(\zeta)$ imposing ζ to have a constant width and yielding its value, as given by Definition 4.2.2. For a rule $\mathrm{T} \in R$, we also define ante(T) and dest(T) to be respectively the antecedent of T and the destination zone of the transition inferred in the consequent of T.

We consider antecedents and destinations, as respectively written in each rule of R, to be functions of the constraints $(\zeta, \sigma, \upsilon)$ appearing in them. We consider Equ_R to be itself a function of many parameters, for which we define a syntactical shortcut $L \triangleq \zeta, \sigma, \upsilon, \xi, \sigma', \upsilon'$. Hence, for some x not in $clocks(\zeta) \cup clocks(\xi)$, we define Equ_R by:

$$
\begin{array}{l}
Equ_R(\mu, t, u, L, \phi_\zeta, \phi_\xi) \triangleq \\
\quad ((\zeta \Rightarrow \sigma) \Leftrightarrow (\xi \Rightarrow \sigma')) \ \wedge \ ((\zeta \Rightarrow \upsilon) \Leftrightarrow (\xi \Rightarrow \upsilon')) \ \wedge \ (\zeta \Rightarrow \phi_\zeta) \ \wedge \ (\xi \Rightarrow \phi_\xi) \wedge \\
\quad ((\zeta^{\nearrow} \wedge \phi_\zeta) \nLeftrightarrow \zeta) \ \wedge \ ((\xi^{\nearrow} \wedge \phi_\xi) \nLeftrightarrow \xi) \Rightarrow (\\
\quad \left(\bigvee_{T \in R} (\mathrm{ante}(T))(\zeta, \sigma, \upsilon) \right) \ \wedge \ (const\_width(\zeta) = const\_width(\xi)) \wedge \\
\quad (\forall\, \mathrm{T} \in R.\ ((\mathrm{ante(T)})(\zeta, \sigma, \upsilon) \Leftrightarrow (\mathrm{ante(T)})(\xi, \sigma', \upsilon')) \wedge \\
\quad\quad ((\mathrm{ante(T)})(\zeta, \sigma, \upsilon) \Rightarrow (\\
\quad\quad\quad ((((\mathrm{dest(T)})(\zeta^{\downarrow x}, \sigma, \upsilon))^{\swarrow} \wedge \zeta^{\nearrow} \wedge \phi_\zeta)^{/x} \Leftrightarrow (((\mathrm{dest(T)})(\xi^{\downarrow x}, \sigma', \upsilon'))^{\swarrow} \wedge \xi^{\nearrow} \wedge \phi_\xi)^{/x}) \wedge \\
\quad\quad\quad \mu : \phi, \top \rhd \quad t\,(((\mathrm{dest(T)})(t, \zeta^{\downarrow x}, \sigma, \upsilon))^{\swarrow} \wedge \zeta^{\nearrow} \wedge \phi_\zeta) \\
\quad\quad\quad\quad\quad = \quad u\,(((\mathrm{dest(T)})(u, \xi^{\downarrow x}, \sigma', \upsilon'))^{\swarrow} \wedge \xi^{\nearrow} \wedge \phi_\xi)))))
\end{array}
$$

We use two sets of rules as value for R in Equ_R. The first ensemble is used in the prefix-related rules: $Tpref \triangleq \{$TSEL, TURG, TELA, TIDL, TUNBS, TUNBU, TMISS, $\mathrm{TREQ}_{max}\}$. There TREQ_{max} is defined as the application of TREQ that yields the maximal time progression: if ξ_{max} is the destination zone reached by applying TREQ_{max}, then for any other ξ reached by applying TREQ, $((\xi_{max}^{\nearrow} \wedge \xi) \Leftrightarrow \bot)$.

The second set simply contains rules for producing transition of *Error* processes: $Terr \triangleq \{\textsc{Terr}_{max}, \textsc{Tnerr}\}$, where $\textsc{Terr}_{max}$ is defined in the same way as $\textsc{Treq}_{max}$.

We also need to deal with possible zeno and deadlocking executions induced by urgency conditions on prefixes. We hence define for some rule T:

$$\psi_T \triangleq \begin{cases} ((\mathrm{dest}(T))(\zeta^{\downarrow x}, \sigma, \upsilon))^{\swarrow} & \text{if } (\mathrm{ante}(T))(\xi, \sigma', \upsilon'), \\ \top & \text{otherwise.} \end{cases}$$

that is used in prefix and error rules to indicate that the inferred judgment discharges any peer antecedent used in an application of the Choice rule from checking bisimilarity up-to infinity. Bisimilarity up-to ψ_T is then sufficient.

We start the examination of Table 4.6 by rules P-Input, C-Input, Output and Tau, that deal with prefixes. Each of them uses two preconditions. The first (reading from the left-to-right) is an inductive condition requiring that bisimilarity of processes $t\,\zeta$ and $u\,\xi$ be provable up-to infinity; this anticipates the future of processes after each of them has accomplished a *discrete* transition. The other precondition consists in requiring an instantiation of condition Equ_R; this properly deals with *timeliness* equivalence. This condition first ensures that both processes can perform the same discrete and time-passing transitions for all clock valuations satisfying the current zone constraint. It also ensures thet both processes may let the same amount of time elapse: if they have outgoing time-passing transitions, then those transitions should let the same time elapse, and they should reach only bisimilar zones.

The Choice rule deserves some additional comments. It follows the general scheme described above to reach completeness when two processes prevent time progression. A new element is however present in that this is done symmetrically for the two antecedent clauses to the rule: we need only to have $t\,\zeta = u\,\xi$ up-to ψ', and $t'\,\zeta = u'\,\xi$ up-to ψ. The rule needs furthermore to take into account the original delays up-to which the equalities $t\,\zeta = u\,\xi$ and $t'\,\zeta = u'\,\xi$ were proved. The rule Conseq-ϕ tells that diminishing the time up-to which an equality is proved by strenghtening ϕ and, contravariantly, enlarging the amount of time one has to explore to prove the equality of a choice by weakening ψ, is safe. The last two rules can be used to partition the state space should it be too coarse-grained, and to reason about zones which constraints can be satisfied by no clock valuation.

The rules of Tables 4.5 and 4.6 are however unable to treat processes with terms that feature parallel composition. As in other π-calculus theories, we provide an expansion theorem that preserves timed late bisimilarity. This theorem allows to suppress the composition operator from any given process with finite term. Its definition is given on Table 4.7.

In this definition, we have used the term

$$\sum_{\pi_i^t \; opp \; \pi_j^u} ([\mu_i^t \wedge \mu_j^u \wedge a_i = b_j][\sigma_i^t \wedge \sigma_j^u, \upsilon_i^t \vee \upsilon_j^u]\tau.\, v_{ij})$$

as a shortcut for

$$\sum_{\pi_i^t \; opp \; \pi_j^u} ([\mu_i^t \wedge \mu_j^u \wedge a_i = b_j][\sigma_i^t \wedge \sigma_j^u, \upsilon_i^t]\tau.\, v_{ij}) + \sum_{\pi_i^t \; opp \; \pi_j^u} ([\mu_i^t \wedge \mu_j^u \wedge a_i = b_j][\sigma_i^t \wedge \sigma_j^u, \upsilon_j^u]\tau.\, v_{ij})\,.$$

$t_{a(c)} \triangleq ([\sigma,\upsilon]a(c).\,t)\quad t_{a(x)} \triangleq ([\sigma,\upsilon]a(x).\,t)\quad t_{\overline{a}m} \triangleq ([\sigma,\upsilon]\overline{a}m.\,t)\quad t_{\tau} \triangleq ([\sigma,\upsilon]\tau.\,t)$
$u_{b(c)} \triangleq ([\sigma',\upsilon']b(c).\,u)\quad u_{b(x)} \triangleq ([\sigma',\upsilon']a(x).\,u)\quad u_{\overline{b}n} \triangleq ([\sigma',\upsilon']\overline{b}n.\,u)\quad u_{\tau} \triangleq ([\sigma',\upsilon']\tau.\,u)$

P-Input $\dfrac{\mu \rhd t\,\zeta = u\,\xi \quad Equ_{Tpref}(\mu, t_{a(c)}, u_{b(c)}, L, \phi_\zeta, \phi_\xi)}{\mu : \phi, \psi_{\mathrm{TMISS}} \rhd t_{a(c)}\,\zeta = u_{b(c)}\,\xi}\ \square$

C-Input $\dfrac{\mu \rhd t\,(\zeta \wedge x \geq 0) = u\,(\xi \wedge x \geq 0) \quad Equ_{Tpref}(\mu, t_{a(x)}, u_{b(x)}, L, \phi_\zeta, \phi_\xi)}{\mu : \top, \psi_{\mathrm{TMISS}} \rhd t_{a(x)}\,\zeta = u_{b(x)}\,\xi}\ \vartriangle$

Output $\dfrac{\mu \rhd t\,\zeta = u\,\xi \quad Equ_{Tpref}(\mu, t_{\overline{a}n}, u_{\overline{b}n}, L, \phi_\zeta, \phi_\xi)}{\mu : \phi, \psi_{\mathrm{TMISS}} \rhd t_{\overline{a}n}\,\zeta = u_{\overline{b}n}\,\xi}\ \triangledown$

Tau $\dfrac{\mu \rhd t\,\zeta = u\,\xi \quad Equ_{Tpref}(\mu, t_{\tau}, u_{\tau}, L, \phi_\zeta, \phi_\xi)}{\mu : \phi, \psi_{\mathrm{TREQ}_{max}} \rhd t_{\tau}\,\zeta = u_{\tau}\,\xi}$

Error $\dfrac{Equ_{Terr}(\mu, Error^{\upsilon}, Error^{\upsilon'}, L, \phi_\zeta, \phi_\xi)}{\mu : \phi, \psi_{\mathrm{TERR}_{max}} \rhd Error^{\upsilon} = Error^{\upsilon'}}$

Reset $\dfrac{\mu : \phi, \psi \rhd t\,\zeta^{\downarrow x} = u\,\xi^{\downarrow x}}{\mu : \phi, \psi \rhd (\nu x\ t)\zeta = (\nu x\ u)\xi}\ x \notin (clocks(\zeta) \cup clocks(\xi))$

Choice $\dfrac{\mu : (\phi \wedge \psi'), \psi \rhd t\,\zeta = u\,\xi \quad \mu : (\phi' \wedge \psi), \psi' \rhd t'\,\zeta = u'\,\xi}{\mu : (\phi \wedge \phi'), (\psi \wedge \psi') \rhd (t + t')\,\zeta = (u + u')\,\xi}$

Conseq-ϕ $\dfrac{\mu : \phi, \psi \rhd t\,\zeta = u\,\xi}{\mu : \phi', \psi' \rhd t\,\zeta = u\,\xi}\ (\phi' \Rightarrow \phi) \wedge (\psi \Rightarrow \psi')$

Part-ζ $\dfrac{\mu : \phi, \psi \rhd t\,\zeta = u\,\xi \quad \mu : \phi, \psi \rhd t\,\zeta = u\,\xi'}{\mu : \phi, \psi \rhd t\,\zeta = u\,(\xi \vee \xi')}\ \blacktriangle$

Absurd-ζ $\dfrac{-}{\mu : \phi, \psi \rhd t \perp = u \perp}$

$\square \triangleq (\mu \Rightarrow (a = b)) \wedge (c \notin ports(\mu)).\quad \phi_\zeta \triangleq \phi^{/clocks(\zeta)}\quad \phi_\xi \triangleq \phi^{/clocks(\xi)}$
$\vartriangle \triangleq (\mu \Rightarrow (a = b))\ \wedge\ (c \notin (clocks(\zeta) \cup clocks(\xi)))\ .$
$\triangledown \triangleq (m \in \mathbf{P} \Leftrightarrow n \in \mathbf{P}) \wedge (m \in \mathbf{P} \Rightarrow (\mu \Rightarrow m = n)) \wedge (\mu \Rightarrow (a = b))\ .$
$\blacktriangle \triangleq (\xi \text{ and } \xi' \text{contiguous}) \wedge (\exists d.\ const\_width(\xi \vee \xi') = d)$

Table 4.6: Symbolic Proof System for π^{δ} : Part 2

For any $t \equiv \nu x\ \sum_i [\mu_i^t][\sigma_i^t, \upsilon_i^t]\pi_i^t.\, t_i$ and $u \equiv \nu x\ \sum_j [\mu_j^u][\sigma_j^u, \upsilon_j^u]\pi_j^u.\, u_j$ with $(\{x\} \cap (\mathrm{fc}(t) \cup \mathrm{fc}(u))) = \emptyset$ we have:

$$\begin{aligned}(t \mid u)\,\zeta \sim_{stl} (\nu x\ (\quad & \sum_i [\mu_i^t][\sigma_i^t, \upsilon_i^t]\pi_i^t.\, (t_i \mid u) \\ + \quad & \sum_j [\mu_j^u][\sigma_j^u, \upsilon_j^u]\pi_j^u.\, (t \mid u_j) \\ + \quad & \sum_{\pi_i^t\ opp\ \pi_j^u} [\mu_i^t \wedge \mu_j^u \wedge a_i = b_j][\sigma_i^t \wedge \sigma_j^u, \upsilon_i^t \vee \upsilon_j^u]\tau.\, v_{ij} \\ + \quad & \sum_i Error^{(\upsilon_i'^{=})'} + \sum_j Error^{(\upsilon_j'^{=})'}))\,\zeta\end{aligned}$$

with $\pi_i^t\ opp\ \pi_j^u$ is true when:

- $\pi_i^t = a_i(m_i)$ for some m_i and $\pi_j^u = \overline{b}_j n_j$ for some n_j; then $v_{ij} \triangleq t_i[n_j/m_i] \mid u_j$,
- $\pi_i^t = a_i(b_i)$ for some b_i and $\pi_j^u = \overline{c}_j(d_j)$ for some d_j; then

  $$v_{ij} \triangleq \nu e\ (t_i[e/b_i] \mid u_j[e/d_j])$$

  for some $e \notin (\mathrm{fn}(t_i) \cup \mathrm{fn}(u_j))$,
- the two above cases may be applied by inverting t with u, and i with j.

Table 4.7: An Expansion Theorem for Finite π^δ Symbolic Terms

We remark that this expansion theorem indeed safely strengthens the one used in π-calculus theories [SW01]. That is, if we take all σ and all υ to be respectively $\top$ and $\bot$, in the communication alternative (the one yielding a τ action) the constraints $\top \wedge \top$ and $\bot \vee \bot$ are then respectively equivalent to $\top$ and $\bot$.

4.2.4 Soundness and Completeness

If there exists a derivation built through our proof systems that concludes $\mu \rhd t\,\zeta = u\,\xi$ for some μ, t, u, ζ, ξ, then we write $\vdash \mu \rhd t\,\zeta = u\,\xi$. We prove a restricted form of soundness and completeness, interesting ourselves only to symbolic timed late bisimilarity (up-to infinity). Our proof system is then *sound* because, for any judgment $\mu \rhd t\,\zeta = u\,\xi$ that can be proved using this system, the processes $t\,\zeta$ and $u\,\xi$ are symbolic timed late bisimilar. It is *complete* because conversely, for any two symbolic timed late bisimilar processes $t\,\zeta$ and $u\,\xi$, the judgment $\mu \rhd t\,\zeta = u\,\xi$ can be derived using the axioms and rules of our proof system.

Theorem 4.2.4 (Soundness of the Proof System).
if $\vdash \mu \rhd t\,\zeta = u\,\xi$, *then* $t\,\zeta \sim^{\mu}_{stl} u\,\xi$.

Theorem 4.2.5 (Completeness of the Proof System).
if $t\,\zeta \sim^{\mu}_{stl} u\,\xi$, *then* $\vdash \mu \rhd t\,\zeta = u\,\xi$.

Both proofs of those theorems use standard induction techniques on the length of the terms and proofs. The proof of soundness is mainly based on Theorem 4.2.1, and it is not very difficult to establish. The proof of completeness is trickier, the core problem being that time-locks may occur in the system. To solve a similar issue, Lin and Yi actually use a *bisimulation up-to* some given deadline, and show that this deadline can grow unboundedly. As we introduced up-to notion directly within our bisimulation we only have to rely on transition induction to settle this part of the work.

4.3 Conclusion

We have in this chapter presented an abstract semantics for terms of the calculus defined in Chapter 3. This abstract semantics is two-folded: it deals with naming aspects vernacular to the π-calculus and with timing aspects present in dense real-time process algebras othogonally. Naming aspects are tackled in a nearly identical way as it had been in the (untimed) π-calculus. This is a conspicuous sign that we have achieved our goal of conservatively extending π-calculus theories. Timing aspects are handled by quotienting the concrete state space through a time-abstracting ground bisimulation. Although the abstract behavior of a process is not a completely safe abstraction of the concrete one, we have been able to devise an abstract notion of bisimilarity that exactly corresponds to the concrete one, and to provide a proof system over finite terms for it.

Our way of performing time abstraction is different from other proposals aiming at axiomatization of timed bisimulation [BD94, LY00]. Those axiomatization indeed rely on the adaptation of the symbolic bisimulation technique to a timed setting,

and they place time-related abstraction information on transitions, not on states. We depart from this because our model does not have the time continuity property (see Section 3.8). This has the consequence that processes reachable from a given process by letting time pass may belong to different time-abstracting equivalence classes, though they have the same term. As symbolic methods assume that all processes with the same term belong to the same equivalence class, we can not apply this approach here.

Our abstract semantics is based on zones instead of regions because zones provide a more abstract way to describe states, hence yielding more succinct and elegant operational rules. However, as zones fail to be concrete enough in the general case, we have introduced two ways of refining the abstract state space. First, the rules as TREG, TREQ, TERR and TNERR state that whenever a region $t\,\zeta$ is reachable, then all the regions between the current one and $t\,\zeta$ are reachable too. Second, the definition for symbolic times late bisimulation allows to partition the state space in order to make the reasoning as accurate as needed.

On the related work side, there are many papers on the axiomatization and decidability of bisimulations and timed for regular processes that we should cite. Since the provision by Milner of a proof system for bisimulation and an axiomatization for observational congruence in the regular subset of CCS [Mil84, Mil89b, Mil89a], many works aimed at devising similar results for the π-calculus. We already mentioned many of them in Section 4.1.2. An essential problem was at first that CCS's ground bisimulation is not a congruence: in the π-calculus, it is not preserved by parallel composition [San93]. Milner, Parrow and Walker hence distinguished the late and early bisimulations [MPW92, MPW93], that are preserved by the parallel operator, but unfortunately fail to be a congruence because they are not preserved by input prefix. This way noticed by Sangiorgi, who proposed open bisimilarity as a (stronger) core equivalence preserved by all π-calculus operators. The axiomatizations and proof systems given in the references above were however complete only at the expense of adding a *mismatch* operator in the calculus. This introduction was controversial, because it leads to "bad" semantical properties [SW01], though at some scarce occasions it also encountered well-founded support [BN95]. The proof systems given in [San93, Qua99, BD94] for this extension of the π-calculus were all based on variants of the distinction-indexed bisimulation proposed in [MPW92]. A proper solution for the mismatch-free calculus was provided in [Lin94].

For timed process algebras, an early complete axiomatization of timed bisimulation for a regular subset of processes written in TCCS [Yi91] for which it is a congruence can be found in [AJ95]. Larsen and Yi introduced time-abstracted bisimulation [HYL92] and proved that it is a congruence regarding parallel composition for the same TCCS. Independently, the track of symbolic bisimulations for timed systems had been pioneered by Boreale [Bor96] until Lin and Yi achieved a complete proof-theoretic characterization of timed bisimulation for timed automata using the same symbolic technique [LY00]. Meanwhile, many authors provided sound but incomplete axioms and proof systems for various notions of equivalence (mainly timed bisimulation) on their timed process algebras [Mol90, HR95, Yi91, Che92, ST92, Mol90, NSY93, DB96, BB91, FK95, BS00]. Kārlis Čerāns [Č92], proved the decidability of timed bisimulation for a variant of timed automata called parallel

timer processes (timers see their values decrease with time progression down to 0, oppositely from clocks, wich value may only increase).

Our work is related to the works mentioned above, but it gives a proof system for models that lack time time continuity property, which is present in all other contributions. This has turned to be somewhat technically challenging. We propose an expansion theorem for terms with name mobility and time, which has not been done before either. Discussions on the existence of such expansion theorems for timed processes can be found in [GL92].

We think that we should finally cite several other works that adopted alternative approaches to the one we chose, but that are also certainly related. First, an other preorder has commonly been used for time systems, based on the relative speed of processes; it is called the *faster-than* preorder. Selected references include [MT91], [GRS95] and [BLS00]. We think that this way of relating processes does not correspond to what is wished for the time-bound reative systems that we aim at representing: a household saying of real-time system engineering tells that those systems have to be *on time*, but they do not necessarily have to be *fast*. Other proof systems have also been proposed, relying on timed temporal logics as specification language. Ostroff [Ost89] on the one side, and Henzinger, Manna and Pnueli [HMP91] on the other have devised proof systems for timed extensions of linear-time temporal logics. Hooman and Widom [HW89, Hoo98] have proposed *compositional* proof systems for timed specification logics; so did Abadi and Lamport [AL94] in the framework of TLA.

As of possible future works, we can name at least to issues that may be worth to tackle. The first is to adapt our bisimulation relations and our proof system to cater with *early* instanciation of port names. The second point is to show how a *mismatching* operator could be introduced. Although we conjecture that the second point can be handled easily following [Lin94], we think that the first point could be trickier. Indeed, our operation semantics handles the transmission of clock values in a *late* fashion: a process considers that the received value can be any positive real number. Our notion of equivalence, following the same trend, hence advises that bisimilar processes should behave in the same way whatever the value of the received clock is. This definitely stands for a late instantiation of received clock values. We do not know how easy or hard adapting the current developments to early instantiation for clock value transmission could be.

Chapter 5

A Behavioral Type System for π^δ

5.1 Introduction

Having a proof system for timed bisimilarity on finite-state processes is nice, but it is not sufficient. Indeed, modeling the simplest problems generally require processes that use recursion, hence outreaching the equivalence-based verification method we proposed earlier. Furthermore, as recursion-featuring processes can have an infinite state-space, they can not accommodate for exhaustive state-space search methods either.

We propose to use a static analysis method based on *behavioral typing*. Originating from the simply typed λ-calculus of Church and Curry [Bar92], program annotations called *types* have lead their way into everyday computer science. Types form a convenient way to indicate the intended use of program constructs, allowing to avoid a large part of the most common programming mistakes during early software development. In our type system, each port is annotated with a higher-order deterministic timed automaton that indicates the rights and obligations of the process that owns this port. A process failing to fulfill the obligations imposed by the types of the ports it owns or overpassing the rights given on the same ports is declared *ill-typed*. A process composition $P_1 \mid P_2$ is well typed only if P_1 and P_2 are well-typed and if they agree *at some level* on the types of their free names. This level of agreement for types is usually called *type compatibility*.

This way, we are able to guarantee the absence of communication errors in well-typed configurations. Our types are *approximations* of the behavior of processes, and a subset of the processes that *do not* exhibit communication errors are however ill-typed. This deficiency can not be avoided, because detecting communication errors in π-calculus terms is in general undecidable [VR99].

We give a type system allowing to infer, for a *typing environment* Γ and a process P, whether the use made by P of the names present in Γ is correct, that is noted as usual $\Gamma \vdash P$. Our type system enjoys the well-known *subject reduction* property, and also accommodates for a notion of *subtyping*. That subtyping notion is based on an asymmetric kind of *simulation* between types (or equivalently asymmetric *language inclusion*, since types are deterministic), while type equivalence is naturally decided by *bisimulation* (or language equivalence).

We use the subject reduction result to prove a *safety* property and a *liveness*

property [AS85]. The safety property is the absence of the *Error* process in the reachability set of a well-typed configuration. The liveness property is that every execution is *non-zeno*: any infinite execution always let time progress up-to infinity. Both results are obtained through a compositional, circular reasoning, called *assume/guarantee* reasoning [MC81, Jon83, BKP84, Sta85, AL95]. It involves proving the soundness of the typing rule for parallel composition $P_1 \mid P_2$ by performing a mutual induction on the length of the executions of P_1 and P_2. Performing assume/guarantee reasoning to prove safety properties is a household method. Solutions dealing with some liveness properties have been proposed [PJ91, AL93], but the problem has only been tackled in a general way quite recently for linear-time specifications, by Kenneth McMillan [McM99]. Our proof of liveness is conditioned by the *strong non-zenoness* of processes [SY96, BGS00, BS00, BST97], and by a weak fairness assumption on the occurrence of silent transitions with urgency.

The remaining of this chapter is organized as follows. First, we introduce the formal syntax and semantics of types in Section 5.2. We also give an informal description on the relation between types and processes in Subsection 5.2.3. In Section 5.3 we define type equivalence and our notion of subtyping. An axiomatic definition of type compatibility is given in Section 5.4, along with the type system. We henceforth give a few theorems about our type system and well typed processes in Section 5.5, extending the range of studied properties of well-typed processes to liveness properties in Section 5.6. We give a rather short comparison to other proof and type systems with our conclusion in Section 5.7.

5.2 The Type Language

In existing π-calculi theories, types can be borne by ports, by processes, or by both of them. In the case of typed processes, a type can be seen as a safe abstraction of the behavior of all processes that have this type [Bou97a, CRR02, Cou97]. In the case of typed ports, a type can be rather seen as an abstract specification describing how the port can or should be used: for example, in the simply typed polyadic π-calculus [Mil93], port types contain the list of typed formal parameters that can be sent through a channel having this type. It has however been found rapidly that the type system proposed in [Mil93] is unsatisfactory: it is unable to detect most inappropriate uses of ports, even in the simplest cases.

A classical problematic example is the *one-place buffer*, that can be written in the following way:

$$Full(elt) \triangleq [\top, \bot]get(reply).\, \nu x\ [x < 1, x < 1]\overline{reply}(elt).\, Empty()$$
$$Empty() \triangleq [\top, \bot]put(elt).\, Full(elt)$$

The buffer starts logically in the state *Empty*. Along its execution, we can see that the buffer shows a *non-uniform* interface to its environment. It means that all environments are not suitable to the buffer processes, and that an inappropriate environment can lead the whole system to produce an error. Precisely, the environment should not impose synchronization on port *get* when the buffer is empty, while it should not impose synchronization on *put* when the buffer is full. Furthermore,

any client should in our case be ready to synchronize on the port *reply* within one time unit after asking the content of the buffer.

The type system of [Mil93] has a very limited power, and can not check the correctness of the above usage. Hence, type systems that denote more complex, *behavioral*, aspects of port usage have been devised [RV97, NN97, Pun97, Yos96], mostly originating in the works of Oscar Nierstrasz [Nie95]. Many other works have followed since, applying to derivatives of the π-calculus (essentially its asynchronous variant, as well as actor-based languages). More details about other behavioral type systems can be found in the *related works* section of this chapter.

But let us now expose our contributions in that field. Our types are attributes of (port or clock) names. Processes have no type *per se*. However, as we wrote in the introduction, the rules of our type system will employ typing environments (also know as *typing contexts*), and we may write sometimes that P *has type* Γ whenever process P is *well typed* in context Γ (*i.e.* $\Gamma \vdash P$).

5.2.1 The Syntax for Types

In our system, a type describes the usage that ought-to be respected for a given port; this description gives the order in which successive interactions can be performed, the nature of those interactions according to the input/output distinction, the type of the object port that will be transmitted when interacting, and the timeliness constraints that relate consecutive interactions.

Types therefore denote timed behaviors and, as processes, use clocks and constraints on the values of those clocks to impose timeliness. As process actions, type actions can become urgent, forcing the action to occur before some deadline deducible from the urgency condition.

As processes, well-formed port types thus have two parts: a term, formed over an algebra that yields only regular processes, and a context part that comprises a value for each free clock name appearing in the corresponding term. There are hence uncountably-many types. To perform type-checking of process configurations, we therefore need an abstraction for types as we needed one for processes; our abstraction operation for types is the quotient of the concrete transition system by a variant of the *time-abstracting bisimulation* seen earlier. Yet, we will not even bother about giving a concrete semantics for types. We define instead our type semantics directly at the abstract level.

We furthermore need to distinguish between *clock types* and *port types*. It must be easy for the reader to accept the following: a clock type is simply a constraint that gives bounds to the clock's value. Syntactically, we range over types with $\mathcal{T}, \mathcal{U}, \mathcal{V} \ldots$ and their decorated variants. We define:

$$\mathcal{T} ::= \zeta \mid T\zeta$$

where T is a behavior type term. For those terms we shall adopt the syntax:

$$T ::= 0 \mid [\sigma, \upsilon]\pi_{\updownarrow}.T \mid \nu x\ T \mid M(\tilde{x}) \text{ where } \pi_{\updownarrow} ::= \uparrow \mathcal{T} \mid \downarrow \mathcal{T}\ .$$

In type actions, $\uparrow$ corresponds to output prefix, while $\downarrow$ corresponds to input prefix. Each type action prefix also encloses the type of the passed name. As in the case

of processes, we assume the existence of a finite set of (potentially recursive) type name definitions, over which we range by $M, N, O \ldots$ Only clocks can be passed as arguments when calling a pre-defined named type, and such a definition is written $M(\tilde{x}) \triangleq T$ where the set of free clocks $\mathrm{fc}(T)$, defined as it was for process terms, is such that: $\mathrm{fc}(T) \subseteq \{\tilde{x}\}$.

A type environment Γ is written in the following way:

$$\Gamma ::= a : T\,\zeta \mid a : * \mid A(\tilde{\mathcal{T}}) \mid \Gamma, \Gamma$$

where a is a port name.

An environment Γ essentially comprises a set of couples $a_1 : T_1\zeta_1, \cdots, a_k : T_k\zeta_k$ associating each port name a_i to a type $T_i\zeta_i$; the *domain* of Γ is defined by $dom(\Gamma) \triangleq \{a_1, \ldots, a_k\}$. We note $\Gamma(a_i) \triangleq T_i\zeta_i$; $\Gamma(a_i)$ is undefined for any $a \notin dom(\Gamma)$. All names in Γ must be pairwise different: $\forall i, j \in \{1..k\}.(i \neq j) \Rightarrow (a_i \neq a_j)$. The purpose of a typing environment Γ is to provide a type for each *free name* appearing in the typed process P: we allow to write $\Gamma \vdash P$ only if $\mathrm{fn}(P) \subseteq dom(\Gamma)$. The special type $*$ means that a is *non-composable*: a is a free name appearing in the typed process P, but no well-typed environment for P is allowed to use a. This is necessary to define the composition rules in Subsection 5.4.3. Typing environments are also used to deal with named (potentially recursive) processes. Named processes are therefore typed separately, their initial environment being defined by the set of received arguments. When typing a given process, the environment must contain an indication that any named process that is called is well-typed, and that the types of the arguments match the types of the formal parameters of the named process.

5.2.2 Type Semantics

We give a semantics to both port types and typing contexts. We chose to impeach actions performed by types of names belonging to the same typing context to interfere with each other. The sought result is that the semantics of a typing context is straightforwardly deduced from the semantics of the types of names in its domain: it is the interleaving of all the type actions that can be performed by each type. But first, we must define the semantics of a type.

The semantics of a type is given by a *higher-order timed transition system* (we shall often leave implicit the "higher-order" qualifier). The labels of the transitions appearing in the system can be written:

$$\lambda_\updownarrow ::= \pi_\updownarrow \mid \langle \pi_\updownarrow^{mode} \rangle \mid \langle\rangle$$

where the mode is defined by $mode ::= o \mid r$ that stand respectively for *offer* and *requirement*. The higher-order aspect is induced by the possibility of making types appear within labels of transitions. Another noticeable difference with the process semantics is that types ignore naming issues: a type does not make use of any port name. This means that our symbolic semantics for types do not have to bear name matching sets.

The semantics of types is given in Tables 5.1 and 5.2. Due to the simplicity of the type terms, many rules that were needed to define the semantics of processes

$$\textsc{Pre}\ \frac{\zeta \Rightarrow \sigma}{([\sigma,\upsilon]\pi_{\updownarrow}.T)\,\zeta \xrightarrow{\pi_{\updownarrow}} T\,\zeta'} \qquad \textsc{Reset}\ \frac{T\,\zeta^{\downarrow x} \xrightarrow{\lambda_{\updownarrow}} T'\,\zeta'}{(\nu x\ T)\,\zeta \xrightarrow{\lambda_{\updownarrow}} T'\,\zeta'}\ \Box$$

$$\textsc{Id}\ \frac{T\,[\tilde{y}/\tilde{x}]\,\zeta \xrightarrow{\lambda_{\updownarrow}} T'\,\zeta'}{(M(\tilde{y}))\,\zeta \xrightarrow{\lambda_{\updownarrow}} T'\,\zeta'}\ M(\tilde{x}) \triangleq T \qquad \textsc{Conv}\ \frac{U\xi \equiv T\zeta \quad T\zeta \xrightarrow{\lambda_{\updownarrow}} T'\zeta' \quad U'\xi' \equiv T'\zeta'}{U\xi \xrightarrow{\lambda_{\updownarrow}} U'\xi'}$$

$$\Box \triangleq (x \notin \mathit{clocks}(\zeta)) \wedge (\mathit{ceil}(x,\zeta^{\downarrow x}) = \mathit{ceil}(x,T))$$

Table 5.1: Type Semantics: Part 1

have been pruned in the type semantics. We use α-conversion and conversion of free clock names as congruence relation $\equiv$ (see the previous Section 4.1.4 for formal definitions).

We shall not give any detailed explanation for the rules shown in Tables 5.1 and 5.2; they are simple adaptations of the corresponding rules for symbolic process semantics. There is however one major difference in the treatment of non-urgent prefixes. The rule TSKIP indeed introduces the ability of *skipping* one or more prefix actions, in case time may pass so that the selection condition of a prefix $[\sigma,\upsilon]\pi_{\updownarrow}.T$ becomes false. This allows a type to change an offer whenever the time period during which it was available has ended up. This corresponds to the fact that an offer made by a process has not been chosen by the environment of that process. The process may then logically change its offer.

The rule TSKIP is certainly useful, but it has a *major* drawback. It allows types to produce *non-deterministic* computations. Suppose indeed that an offer ends-up when the condition $x \leq 4$ on some clock x becomes false. If the continuation of the corresponding prefix starts by another prefix of the form $[4 \leq x, \bot]\pi'_{\updownarrow}.T'$, then two (potentially different) offers are available at the instant when $x = 4$. Since our type system may accommodate for deterministic types only, we have to prevent such situations. The problem of telling whether a type yields a non-deterministic transition system is statically decidable, since the state-space of our processes is finite. One could however devise some (stronger) syntactic constraints ensuring the very same fact. Since there can be many such syntactic constraints, we will not elaborate and let the reader pick up one he or she thinks appropriate. We shall simply assume from now on that types yield deterministic transition systems.

Using the semantics of types, we can give a semantics to type environments Γ. We use a very simple congruence relation, that establishes commutativity for the comma operator ($\Gamma, \Gamma' \equiv \Gamma', \Gamma$) and extends the type congruence:

$$((T\zeta \equiv T'\zeta') \wedge (\Gamma \equiv \Gamma')) \Rightarrow (\Gamma, a : T\zeta \equiv \Gamma', a : T'\zeta')\ .$$

Typing context semantics is quite simple. First, atoms consisting of named processes or uncomposable ports are taken away from the context; they perform no action, staying unchanged under any context transition. For the other atoms in the

$$\textsc{Tsel}\ \frac{((\zeta \wedge \sigma^{\uparrow}) \Leftrightarrow \bot) \wedge (\zeta^{\uparrow\downarrow} \Rightarrow \sigma^{\uparrow\downarrow})}{([\sigma,\upsilon]\pi_{\updownarrow}.T)\,\zeta \xrightarrow{\langle\rangle} ([\sigma,\upsilon]\pi_{\updownarrow}.T)(\sigma \wedge \zeta^{\uparrow})^{=\downarrow}}$$

$$\textsc{Turg}\ \frac{(\zeta \Rightarrow (\sigma \backslash \upsilon^{\uparrow})) \wedge (\zeta^{\uparrow\downarrow} \Rightarrow \upsilon^{\uparrow\downarrow})}{([\sigma,\upsilon]\pi_{\updownarrow}.T)\,\zeta \xrightarrow{\langle\pi^o_{\updownarrow}\rangle} ([\sigma,\upsilon]\pi_{\updownarrow}.T)(\upsilon \wedge \zeta^{\uparrow})^{=\downarrow}}$$

$$\textsc{Tela}\ \frac{(\zeta \Rightarrow (\sigma \backslash \upsilon^{\uparrow})) \wedge (\sigma^{\uparrow} \not\Leftrightarrow \sigma) \wedge ((\upsilon \wedge \zeta^{\uparrow}) \Leftrightarrow \bot)}{([\sigma,\upsilon]\pi_{\updownarrow}.T)\,\zeta \xrightarrow{\langle\pi^o_{\updownarrow}\rangle} ([\sigma,\upsilon]\pi_{\updownarrow}.T)(\sigma \wedge \zeta^{\uparrow})^{\uparrow=}}$$

$$\textsc{Tskip}\ \frac{T\zeta \xrightarrow{\lambda_{\updownarrow}} T'\zeta' \quad (\sigma^{\uparrow} \not\Leftrightarrow \sigma) \wedge (\zeta \Rightarrow \sigma^{\uparrow=})}{([\sigma,\upsilon]\pi_{\updownarrow}.T)\,\zeta \xrightarrow{\lambda_{\updownarrow}} T\,\zeta'}$$

$$\textsc{Tunbs}\ \frac{(\zeta \Rightarrow \sigma) \wedge (\sigma^{\uparrow} \Leftrightarrow \sigma) \wedge ((\upsilon \wedge \zeta^{\uparrow}) \Leftrightarrow \bot)}{([\sigma,\upsilon]\pi_{\updownarrow}.T)\,\zeta \xrightarrow{\langle\pi^o_{\updownarrow}\rangle} ([\sigma,\upsilon]\pi.T)\zeta^{ceil}}$$

$$\textsc{Tunbu}\ \frac{(\zeta \Rightarrow \upsilon) \wedge (\upsilon^{\uparrow} \Leftrightarrow \upsilon)}{([\sigma,\upsilon]\pi_{\updownarrow}.T)\,\zeta \xrightarrow{\langle\pi^r_{\updownarrow}\rangle} ([\sigma,\upsilon]\pi_{\updownarrow}.T)\,\zeta^{ceil}}$$

$$\textsc{Treg}\ \frac{([\sigma,\upsilon]\pi_{\updownarrow}.T)\,\zeta \xrightarrow{\langle\pi^{mode}_{\updownarrow}\rangle} ([\sigma,\upsilon]\pi_{\updownarrow}.T)\,\zeta' \quad (\xi \Rightarrow (\zeta^{\uparrow} \backslash \zeta^{=\downarrow}) \backslash \zeta'^{\uparrow}) \wedge (\xi^{\uparrow\downarrow} \Leftrightarrow \zeta^{\uparrow\downarrow})}{([\sigma,\upsilon]\pi_{\updownarrow}.T)\,\zeta \xrightarrow{\langle\pi^{mode}_{\updownarrow}\rangle} ([\sigma,\upsilon]\pi_{\updownarrow}.T)\,\xi}$$

$$\textsc{Treq}\ \frac{(\zeta \Rightarrow \upsilon) \wedge (\upsilon^{\uparrow} \not\Leftrightarrow \upsilon) \wedge (\xi \Rightarrow (\upsilon \wedge (\zeta^{\uparrow} \backslash \zeta^{=\downarrow}))) \wedge (\xi^{\uparrow\downarrow} \Leftrightarrow \zeta^{\uparrow\downarrow})}{([\sigma,\upsilon]\pi_{\updownarrow}.T)\,\zeta \xrightarrow{\langle\pi^r_{\updownarrow}\rangle} ([\sigma,\upsilon]\pi_{\updownarrow}.T)\xi}$$

Table 5.2: Type Semantics: Part 2

context, discrete actions of separate types are interleaved (there is no possibility to synchronize discrete actions), while time must pass in a uniform fashion for all types in the context. The labels of typing context transitions range over

$$\lambda_\Gamma ::= a(\mathcal{T}) \mid \overline{a}(\mathcal{T}) \mid \langle \mathcal{R}_\Gamma, \mathcal{O}_\Gamma \rangle$$

where $\mathcal{R}_\Gamma$ and $\mathcal{O}_\Gamma$ are the requirement and offer sets, containing elements of the form $a(\mathcal{T})$ or $\overline{a}(\mathcal{T})$.

There are only two semantic rules for typing contexts:

$$\text{DISC}\ \frac{T\zeta \xrightarrow{\pi_\updownarrow} T'\zeta'}{\Gamma, a : T\zeta \xrightarrow{a\pi_\updownarrow} \Gamma, a : T'\zeta'}$$

$$\text{TIMED}\ \frac{\begin{array}{c} T_1\,\zeta_1^{\downarrow x} \xrightarrow{\langle \pi_{1\updownarrow}^{mode_1} \rangle} T_1'\,\zeta_1' \\ \vdots \\ T_k\,\zeta_k^{\downarrow x} \xrightarrow{\langle \pi_{k\updownarrow}^{mode_k} \rangle} T_k'\,\zeta_k' \\ \forall i, j \in \{1..k\}.\ \zeta_i'^{/x} \Leftrightarrow \zeta_j'^{/x} \\ \Gamma \equiv a_1 : T_1\,\zeta_1, \ldots, a_k : T_k\,\zeta_k \end{array}}{\Gamma \xrightarrow{\uplus(a_1\pi_{1\updownarrow}^{mode_1}, \ldots, a_k\pi_{k\updownarrow}^{mode_k})} a_1 : T_1'\,\zeta_1',\ \ldots, a_k : T_k'\,\zeta_k'}$$

Both rules use the name prefixing operator on type action prefixes that, for any port name a and prefix $\pi_\updownarrow$, we define as: $a\pi_\updownarrow \triangleq \overline{a}(\mathcal{T})$ if $\pi_\updownarrow = \uparrow \mathcal{T}$ and $a \downarrow \triangleq a(\mathcal{T})$ if $\pi_\updownarrow = \downarrow \mathcal{T}$. The rule for time progress furthermore uses a "union sum" operator defined inductively on lists of type actions by: $\uplus(\emptyset) = \langle \emptyset, \emptyset \rangle$, and with $head(S) = a\pi_\updownarrow^{mode}$, $\uplus(S) = \langle a\pi_\updownarrow : \emptyset \rangle \cup (\uplus(tail(S)))$ if $mode = r$, and $\uplus(S) = \langle \emptyset : a\pi_\updownarrow \rangle \cup (\uplus(tail(S)))$ if $mode = o$. The union of offer/requirement sets is defined as in Section 3.6: $\langle \mathcal{R}_\Gamma, \mathcal{O}_\Gamma \rangle \cup \langle \mathcal{R}'_\Gamma, \mathcal{O}'_\Gamma \rangle = \langle \mathcal{R}_\Gamma \cup \mathcal{R}'_\Gamma, \mathcal{O}_\Gamma \cup \mathcal{O}'_\Gamma \rangle$. Furthermore, in the TIMED rule, we left implicit that some transition labels $\langle \pi_{k\updownarrow}^{mode_k} \rangle$ may actually be $\langle\rangle$. This rule of course also apply to those empty action sets, leading to empty offer/requirement sets $\langle \emptyset : \emptyset \rangle$.

We extend the usual operations on clock constraints to typing contexts. Whenever $\Gamma \equiv a_1 : T_1\,\zeta_1, \ldots, a_k : T_k\,\zeta_k$, we define

$$\begin{array}{l} \Gamma^{\downarrow x} \triangleq a_1 : T_1\,\zeta_1^{\downarrow x}, \ldots, a_k : T_k\,\zeta_k^{\downarrow x} \\ \Gamma^{/x} \triangleq a_1 : T_1\,\zeta_1^{/x}, \ldots, a_k : T_k\,\zeta_k^{/x} \\ \Gamma^{\backslash x} \triangleq a_1 : T_1\,\zeta_1^{\backslash x}, \ldots, a_k : T_k\,\zeta_k^{\backslash x} \\ \Gamma^{\nearrow} \triangleq a_1 : T_1\,\zeta_1^{\nearrow}, \ldots, a_k : T_k\,\zeta_k^{\nearrow} \\ \Gamma^{\swarrow} \triangleq a_1 : T_1\,\zeta_1^{\swarrow}, \ldots, a_k : T_k\,\zeta_k^{\swarrow} \end{array}$$

5.2.3 Discourse on The Meaning of Types

Now that we have defined a proper semantics for types an typing contexts, we may address the issues pertaining to the nature of relations between types and processes. First of all, it is clear that those relations are built on *semantic* grounds. This

is common to all type systems based on the "behaviors-as-types" paradigm. It is however mostly problematic to devise a decidable type system in this context.

Processes in π^δ have the ability to express their needs in terms of synchronization by ensuring themselves of the control of certain prefix actions through the use of urgency conditions. This yields an additional power of scrutiny over process actions: a non-urgent action is an *offer*, while an urgent action is a *requirement*. Processes themsleves enjoy this power, by being able to tell whether their environment is responsive to all their requirements or not. A natural approach to behavioral typing in this context is therefore to let a type reflect the relations a process may have with its environment in terms of offers and requirements.

More precisely, we shall interpret offers and requirements in terms of *assumptions* and *guarantees* that can be respectively made or given by a process to its environment. For a process and its environment to be well-behaved regarding one another, it is important that their assumptions and guarantees match. A process exhibiting an urgent action makes an assumption on its environment, that the environment will provide a matching offer in time. Conversely, a process exhibiting a non-urgent action makes an offer to its environment, so that one process in this environment may force the interaction to occur by imposing an urgency condition on its prefix action.

If a typing context $\Gamma, a : T\zeta$ may let time pass while *offering* to synchronize on a (the label of the action is $\langle a\pi^o_\updownarrow \rangle$ for some $\pi_\updownarrow$), then we consider the context to specify a *guarantee* given by all processes P satisfying $\Gamma, a : T\zeta \vdash P$. Hence, any such P *must* offer to synchronize on a at all the instants when $\Gamma, a : T\zeta$ specifies to do so; the guarantees of P must *exceed* the guarantees of $\Gamma, a : T\zeta$ in order to have $\Gamma, a : T\zeta \vdash P$.

Now consider a *requirement* specified by $\Gamma, a : T\zeta$, that is a transition $\langle a\pi^r_\updownarrow \rangle$ for some $\pi_\updownarrow$. To be well-typed in that context, a process P should exhibit only a *lower* requirement to interact on a: it *may* not require synchronization on a at all, or if requires it, it should let more opportunities to its environment to satisfy that requirement. Hence, it should require to synchronize on a through an urgent action at all times when the typing context specifies to. This corresponds to making lower *assumptions* about the behavior of the environment.

This can be summarized through the following "rule of the thumb":

> *The offers of a process must be* higher *than the offers of its type. The requirements of a process must be* lower *than the requirements of its type.*

This relation between types and processes shall be formalized in Section 5.4, where a set of typing rules is given. This system unambiguously defines when it is possible to prove that $\Gamma \vdash P$ for some Γ and P.

Following the above definitions, we remark that, although type semantics are defined using timed transition systems, which are *branching time* representations of behaviors, those types are only able to represent *linear time* properties. Indeed, types produce only *deterministic* transition systems, and on deterministic transition systems, bisimulation equivalences collapses to language equivalence [vG01]. This categorizes the properties addressed in our results: no properties belonging to the branching time world are considered. Therefore, the proved properties are close

to the ones that could be obtained by assume/guarantee reasoning in the timed versions of some linear time temporal logic (such as MITL [AFH91]), but certainly not branching time temporal logic (such as TCTL [ACD93]).

5.3 Type Equivalence, Subtyping, and Polymorphism

In this section we define a notion of *type equivalence*, as a natural byproduct of a *subtyping* relation. The subtyping relation is useful to perform *hierarchical* development: the implementation can be obtained through a succession of step-wise refinements individually applied to sub-parts of the developed application. Ideally, the final result should still be in conformance with its specification.

A certain form of subtyping can be used in that context. The subtyping relation is very similar to the relatioin between types and processes. We define that a type $U\xi$ is a subtype of a type $T\zeta$ if and only if $U\xi$ has *more offers* but *less requirements* than $T\zeta$. A process Q that satisfies $Q \vdash a : U\xi$ for some a can the replace a process P that satisfies $P \vdash a : T\zeta$ without the environment of P noticing a change in the offers and requirements exhibited by the object. This is often called the *substitution principle* [LW94, WZ88]. This is an application of *inclusion* polymorphism, a form of *universal polymorphism* (for these two notions, see [CW85]). In universal polymorphism, a given value (in our case a port name or a process) can have many types. If those types may include one another, then we obtain inclusion polymorphism. Our subtyping relation provides such an inclusion relation among types.

As the subtyping relation is asymmetric in regards to offers and requirements, we found that a co-inductive definition of subtyping through some sort of simulation would be uneasy, and we prefer to give an axiomatic definition for it. We suppose that types have been put under *one-step normal form*, each named type consisting of a clock reset, a single action, and either a call to another named type, or 0. In the former case, a named type $M(\tilde{x})$ is written: $\nu y\ [\sigma, \upsilon]\pi_{\updownarrow}.N(\tilde{z})$ where $\tilde{z} \subseteq (\{y\} \cup set(\tilde{x}))$. Transforming type terms in one-step normal form can be done trivially by suppressing redundant clock reset with the rule $(\nu x\ \nu y\ T)\zeta \equiv (\nu x\ (T[x/y]))\zeta$, by adding a clock reset wherever one is missing with the rule $([\sigma, \upsilon]\pi_{\updownarrow}.T)\zeta \equiv (\nu x\ [\sigma, \upsilon]\pi_{\updownarrow}.T)\zeta$ that can be applied if $x \notin clocks(\zeta)$, and by taking any prefixed continuation T of a term and assigning to it a name along with a set of parameters and then replacing all its occurrences by a call to the newly defined name.

We then obtain the rather simple rules gathered in Table 5.3. The subtyping relation is the least reflexive and transitive relation closed under those rules. To deal with recursion, we complete each subtyping judgment $T\zeta \preceq U\xi$ with an environment that saves the subtyping judgments that are to be proved by the currently ongoing derivation. This environment therefore consists of pairs $((M_i(\tilde{x}_i))\zeta_i, (N_i(\tilde{x}'_i))\xi_i)$, each member of a pair containing a named process type term and a constraint on the values of its clocks. We note such set of pairs E, and write it at the left of a turnstile, as in: $E \vdash T\zeta \preceq U\xi$. If E is empty, we simply forget the environment, and write the above judgment $T\zeta \preceq U\xi$.

As in the case of our proof system of Section 4, we use a predicate on the possible transitions in order to judge subtyping in a given configuration. The predicate Equ_R

Idle $\dfrac{-}{E \vdash 0\,\zeta \preceq 0\,\xi}$

Absurd-ζ $\dfrac{-}{E \vdash T\bot \preceq U\bot}$

Defn $\dfrac{E, (M(\tilde{x}))\zeta, (N(\tilde{x}'))\xi \vdash (T[\tilde{x}/\tilde{y}])\zeta^{/\tilde{x}} \preceq (U[\tilde{x}'/\tilde{y}'])\xi^{/\tilde{x}'}}{E \vdash (M(\tilde{x}))\zeta \preceq (N(\tilde{x}'))\xi}$ $M(\tilde{y}) \triangleq T$ and $N(\tilde{y}') \triangleq U$

Rec $\dfrac{-}{E, (M(\tilde{x}))\zeta, (N(\tilde{x}'))\xi \vdash (M(\tilde{x}))\zeta \preceq (N(\tilde{x}'))\xi}$

Reset $\dfrac{E \vdash (T\,\zeta^{\downarrow x} = U\,\xi^{\downarrow x})}{E \vdash (\nu x\ T)\zeta \preceq ((\nu x\ U)\xi)}$

Pref $\dfrac{\begin{array}{l} Sub_{Ttype}(\zeta, \sigma, \upsilon, \xi, \sigma', \upsilon') \\ Next(E, T, \zeta, U, \xi) \end{array}}{E \vdash T\zeta \preceq U\xi}$ $T \triangleq [\sigma, \upsilon]\pi_\updownarrow.\, M(\tilde{x})$ and $U \triangleq [\sigma', \upsilon']\pi'_\updownarrow.\, N(\tilde{x}')$

Part-ζ-r $\dfrac{T\zeta \preceq U\xi \quad T\zeta \preceq U\xi'}{T\zeta \preceq U(\xi \vee \xi')}$ Part-ζ-l $\dfrac{T\zeta \preceq U\xi \quad T\zeta' \preceq U\xi}{T(\zeta \vee \zeta') \preceq U\xi}$

Table 5.3: Axiomatic Definition of Subtyping

that we used then has now to be modified in order to take account of the asymmetry of the subtyping relation. We hence define Sub_{Ttype}, defining $Ttype$ as the set {TSEL, TURG, TELA, TSKIP, TUNBS, TUNBU, TREQ_{max}}.

We again take source and destination zones as written in each rule R of $Ttype$, to be the functions ante(R) and dest(R) of constraints $(\zeta, \sigma, \upsilon)$ appearing in them. For some x not in $clocks(\zeta) \cup clocks(\xi)$, we define Sub_{Ttype} by:

$$
\begin{array}{l}
Sub_{Ttype}(\zeta, \sigma, \upsilon, \pi_\updownarrow, \xi, \sigma', \upsilon', \pi'_\updownarrow) \triangleq \\
\left(\bigvee_{R \in Ttype} (\text{ante}(R))(\zeta, \sigma, \upsilon)\right) \wedge \left(\bigvee_{R \in Ttype} (\text{ante}(R))(\xi, \sigma', \upsilon')\right) \wedge \\
((\xi^\uparrow \wedge \sigma' \not\Leftrightarrow \bot) \Rightarrow ((\pi_\updownarrow \preceq \pi'_\updownarrow) \Rightarrow (((\zeta^\uparrow \wedge \sigma) \not\Leftrightarrow \bot) \wedge (((\xi^{\downarrow x})^\uparrow \wedge \sigma')^{/x} \Rightarrow ((\zeta^{\downarrow x})^\uparrow \wedge \sigma)^{/x}))) \wedge \\
\qquad ((\pi_\updownarrow \not\preceq \pi'_\updownarrow) \Rightarrow ((((\xi^{\downarrow x})^\uparrow \wedge \sigma') \wedge ((\zeta^{\downarrow x})^\uparrow \wedge \sigma)^\uparrow) \Leftrightarrow \bot))) \wedge \\
((\zeta^\uparrow \wedge \upsilon \not\Leftrightarrow \bot) \Rightarrow ((\pi_\updownarrow \preceq \pi'_\updownarrow) \Rightarrow (((\xi^\uparrow \wedge \upsilon') \not\Leftrightarrow \bot) \wedge ((\xi^{\downarrow x})^\uparrow \wedge \upsilon')^{/x} \Rightarrow ((\zeta^{\downarrow x})^\uparrow \wedge \upsilon)^{/x})) \wedge \\
\qquad ((\pi_\updownarrow \not\preceq \pi'_\updownarrow) \Rightarrow ((((\zeta^{\downarrow x})^\uparrow \wedge \upsilon) \wedge ((\xi^{\downarrow x})^\uparrow \wedge \sigma')^\uparrow) \Leftrightarrow \bot)))
\end{array}
$$

where the subtyping relation for prefixes $\pi_\updownarrow \preceq \pi'_\updownarrow$ is so that the prefixes have the

same role (sender or receiver) and the type of the argument evolves contravariantly to the subtyping relation itself.

Definition 5.3.1. *We write that $\pi_\updownarrow \preceq \pi'_\updownarrow$ if and only if:*

- *if $\pi_\updownarrow = \uparrow \mathcal{T}$ then $\pi'_\updownarrow = \uparrow \mathcal{T}'$ for some $\mathcal{T}'$ such that $\mathcal{T}' \preceq \mathcal{T}$, and*
- *if $\pi_\updownarrow = \downarrow \mathcal{T}$ then $\pi'_\updownarrow = \downarrow \mathcal{T}'$ for some $\mathcal{T}'$ such that $\mathcal{T}' \preceq \mathcal{T}$.*

The subtyping predicate ensures several points. First, it checks that both types are in an equivalence zone, meaning that one of the antecedent of the rules *Ttype* is true. This is a mandatory condition, since without it the system is conspicuously unsound. Then, two cases may appear. If the supertype will offer an action $\pi'_\updownarrow$ at some time in the future, then the subtype must offer it also: either the prefix $\pi_\updownarrow$ is not a subtype of $\pi'_\updownarrow$ and its availability interval will end before $\pi'_\updownarrow$ availability interval will start, meaning that another subsequent prefix obtained by application of rule TSKIP may be able to propose the demanded offer, or $\pi_\updownarrow$ is a subtype of $\pi'_\updownarrow$, and in that case the subtype must offer this action for a wider interval than the supertype. The converse reasoning is applied to requirement prefixes.

This predicate seems to indeed ensure type safety, but it also seems too permissive on certain aspects, and it obviously leads to *paradoxical* types. Those are types where proving $T\zeta \preceq U\xi$, one needs to prove $T\zeta \not\preceq U\xi$, and vice-versa. Of course, paradoxical types should be banned from correct computations. Detecting them is not very difficult, one having merely to maintain two sets along the subtype-checking procedure, one for subtyping constraints, and the other for negative subtyping constraints. If a couple of types is found to belong to both sets, the procedure is stopped and yields an undecidability result.

If we take a look at the rules of Table 5.3, we encounter the usual rules for idle behavior and anti-logic clock constraints. The rules Defn and Rec are used to perform name invocation and to solve recursive invocations, respectively. The Reset rule allows to introduce clock resets, while the two Part rules at the bottom of the table allow to refine the state space whenever necessary. The rule Pref deals with time-passing and discrete behaviors of prefixed processes. The antecedent occurrence of predicate Sub'_{Ttype} imposes that at any step, the future offers of the supertype be included in the ones of the subtype, and the converse for requirements. Thus, the other predicate $Next$ simply imposes that the types still be in the subtyping relation whatever their evolution can be. This is similar to what is found in simulation relations. if we use the subtyping relation Sub, then $Next$ can be defined by:

$$
\begin{aligned}
&Next(E, T, \zeta, U, \xi) \triangleq \\
&\quad ((T\zeta^{\downarrow z} \xrightarrow{\langle \pi_\updownarrow \rangle} T'\zeta' \wedge U\xi^{\downarrow z} \xrightarrow{\langle \pi'_\updownarrow \rangle} U'\xi' \wedge (\zeta'^{/z} \Leftrightarrow \xi'^{/z})) \Rightarrow \\
&\qquad\qquad (E \vdash (M(\tilde{x}))\zeta'^{\setminus z} \preceq (N(\tilde{x}'))\xi'^{\setminus z})) \wedge \\
&\quad (T\zeta \xrightarrow{\pi_\updownarrow} T'\zeta' \Rightarrow E \vdash T'\zeta' \preceq U\xi) \ \wedge \ (U\xi \xrightarrow{\pi'_\updownarrow} U'\xi' \Rightarrow E \vdash T\zeta \preceq U'\xi')
\end{aligned}
$$

$Next$ imposes that types be synchronized during their time-progressing phases, while they evolve independently when they perform discrete transitions. Like we said, this notion of subtyping is too loose, although it is already *very* restrictive. There is no

doubt in our mind that expressiveness of our processes and type languages is at the source of those restriction. One could probably devise more tolerant, tractable subtyping relations when considering only subset of our language; for example, one may consider that only urgent actions may be refined, leaving identical the offer set of types, or that only output actions may be urgent, etc.

Notice also that in our situation, the substitution principle is unable to guarantee that a correct implementation is obtained after some refinement has been performed. Indeed, our types are too blind observers, and they can not (even remotely) tackle problems related to name equality, for example. This means that, among the processes that possess a certain type (and there are many of them), some will give proper results when put in a given environment, while others will not. Unfortunately, such processes may be indistinguishable in our type system. Subtyping hence only provides a somewhat weak notion of *interface refinement*. To have a general notion of subtyping, we still need to extend it to typing context, that we do now.

Definition 5.3.2 (Context Subtyping). *We write* $\Gamma' \preceq \Gamma$ *for two typing contexts* $\Gamma \equiv a_1 : T_1\,\zeta_1, \ldots, a_k : T_k\,\zeta_k$ *and* $\Gamma' \equiv a_1 : T'_1\,\zeta'_1, \ldots, a_k : T'_k\,\zeta'_k$ *if and only if* $T'_1\,\zeta'_1 \preceq T_1\,\zeta_1, \ldots, T_k\,\zeta_k \preceq T'_k\,\zeta'_k$. *We left aside in* Γ *and* Γ' *the sets of named process types and uncomposable types. The set of named process types of the supertype must be included in the corresponding set of the subtype, while the set of uncomposable types of the supertype and the subtype must be equal.*

We remark on the definition above that object (*i.e.* parameter in action prefix) types evolve *contravariantly* to subject types. Parameter type contravariance is used in all object-oriented formalisms, since "contravariance is safe". It is unfortunately counter-intuitive very often. We remark also that the domain of typing contexts may not vary when they are related by subtyping. This is because, using additional names in the subtype, a process could constrain its environment *more* than in the supertype. The only possibility would be to add names with types that *do not* allow a well-typed process to constrain its environment at all.

We finally define *type equivalence*:

Definition 5.3.3 (Type Equivalence). *Two types* $\mathcal{T}$ *and* $\mathcal{U}$ *are equivalent, noted* $\mathcal{T} \sim \mathcal{U}$, *if and only if* $\mathcal{T} \preceq \mathcal{U}$ *and* $\mathcal{U} \preceq \mathcal{T}$. *In the same way, two typing contexts* Γ *and* Γ' *are equivalent, noted* $\Gamma \sim \Gamma'$, *if and only if* $\Gamma \preceq \Gamma'$ *and* $\Gamma' \preceq \Gamma$.

We will not elaborate much on the decidability of neither the subtyping relation nor the type equivalence. We already saw that our notion of subtyping may lead to inconsistent proofs due to the recursive testing of the subtyping relation in predicate Sub_{Ttype}. We did not explore the syntactic restrictions one could adopt to make subtyping decidable in general. The difficulty of defining such criterion is to keep as much expressiveness as possible, since subtyping is already very restrictive, while allowing an easier subtyping check. Finally, let us just mention that defining subtyping relation is in general difficult, and that the most closest work to ours does not provide a proof of decidability for subtyping or type equivalence [Kob00, Kob02].

5.4 Type Checking

5.4.1 Restraining the Expressive Power of Processes

The type checking problem is in general insolvable for π^δ, since the *no-communication-error* problem (also called *no-missed-synchronization*) is insolvable [VR99]. Hence we have to introduce (strong) restrictions on the sets of processes that may be typed.

The main limitations concern the number of processes that may know a given port at the same time, and the communication of port names among processes. We distinguish between two modes for port manipulation: *public* and *private* [Nie95, NN97].

If a port is public, its use is restricted to either *input* actions or *output* actions, and the occurrences of such actions may not depend on time constraints (the service offered on the port is said *uniform*). A similar restriction has been proposed before by Pierce and Sangiorgi [PS93]. When it may only perform inputs, a public port is said to play the *server* role in the interactions that may occur, while in the other case it is said to play the *client* role. In a process configuration, there many be any number of server roles for a given port, but only one server role. This is the *unique receptor* property, that have been adopted in programming languages such as Pict [PT00] and JoCaml, issued from the Join calculus [FGL+96]. If a client port is sent as object during an interaction, both the sender and the receiver have a copy of the client port after having interacted. If a server role is sent away, then the process that sent it loses the capability to use the port; it may recover this capability by receiving the server role through a posterior communication (this may of course not happen at all).

If a port is private, then the description of the service offered on it may use the full expressive power of our types. However, there may be at one moment only two copies of the name of each port in the system, and the types of those copies must be *compatible*. This notion of compatibility shall be defined formally in the next subsection. To ensure that only two copies of the same private type name are present in the system and compatible, we need to restrain the creation of private names to *bound output* prefixes. This means that a private name can not be created using the general restriction operator, because it could then be shared by many sub-processes put in parallel. Instead, using bound output imposes both names to be created simultaneously, while the compatibility of their types can be checked. In practice, this means that private names can not be shared by many processes in some initial configuration, only public names can. However, private names may be received by open processes during the computation by the way of some free public name they know.

There are also syntactic modifications that must be brought to processes, since we rely on an *explicit* typing paradigm (*à la Church*), and *not* on *implicit* typing (*à la Curry*). Therefore, certain occurrences of port and clock names in processes must bear typing annotations in order to be checkable by our type system. Hence, we give the following syntax for process terms:

$$t ::= 0 \mid Error^{\upsilon} \mid [\sigma,\, \upsilon]\pi.\, t \mid [a = b]t \mid \nu a^{T}\ t \mid \nu x\ t \mid t + t \mid t|t \mid A(\tilde{n})$$

where we again assume the existence of a finite set of named behaviors ranged over

$A, B, C, \cdots$ which syntax for definition is for example $A(\tilde{n} : \tilde{\mathcal{T}}) \triangleq t$ with $\mathrm{fn}(t) \subseteq \{\tilde{n}\}$. We take $\tilde{n} : \tilde{\mathcal{T}}$ to be standing for $n_1 : \mathcal{T}_1, \ldots, n_k : \mathcal{T}_k$.

The name instantiation operator and the prefix operator are modified to enclose typing annotations. Name instantiation only specifies a type for port names, since a clock always evaluates to 0 when it is created. In port name instantiation, we also impose by type checking that the port must be *public*, its role being either client or server. Prefixes are augmented with the *bound output* prefix, that is used to establish communication relying on non-uniform services:

$$\pi ::= a(n^{\mathcal{T}}) \mid \overline{a}\, n \mid \overline{a}(b^{T,U}) \ .$$

The reception of a name includes a type annotation that describes the intended use of the to-be-received name. If the name is a clock, then its type is a constraint on its value. If it is a port, its type is a type term together with a zone constraint. The free output is unchanged, the type of the transmitted name being declared in the instantiation instruction binding that name. The added bound output is somewhat special: it contains two type annotations. This is because the intended meaning of this construct is to establish a session ruling a non-uniform service access (*binding* the sender and the receiver in the sense of ODP, see Subsection 2.1.2.3). The intended meaning of bound output is the following:

$$\overline{a}(b^{T,U}) \equiv \nu b^T \ \nu c^U \ \overline{a}c.\, t$$

where $c \notin \mathrm{fc}(t)$. This corresponds to creating both ends of the binding, and to send one of them. When the synchronization occurs, the clocks of the types start evolving. Hence, as in the name instantiation for public ports, we do not need to adjoin a clock constraint to the given type term to form a full-fledged type: any constraint can actually be adjoined to the type term, since types terms are clock-closed. We suppose that implicitly the empty clock constraint, that we shall note with a pair of empty parentheses () from now on, is adjoined. The moment when the interaction occurs can be used as a synchronization point, that shall be referred to as the time of establishment of the non-uniform service. Although the congruence rule above clearly defines the bound output operation, we retrain from applying it in practice. This is because having different syntactical constructs for uniform and non-uniform service creation allows us to provide a much more simple type system.

In spite of the above modifications, the semantics of our processes is not modified. The semantics of an annotated process is the semantics of the same process after deletion of all type annotations. This means in particular that no error is produced by a process in case of simple ill-typing sources such as transmitting an argument of some private type when a public type is expected. We avoid those kind of errors by type-checking. We conjecture that the expressive power of the language is globally not stifled by the restriction we impose. In particular, techniques for encoding λ-calculus into the π-calculus may still apply. The typed processes hence are still able to generate infinitely-wide state spaces. It is oppositely clear that the synchronization power of processes is greatly reduced, but we do not know how much. Elements on the classification of power of synchronization among concurrent processes have been proposed by Wegner and Goldin [WG02].

Finally, we need a way of representing public (uniform) port types. We will not give any particular syntax to form these public types. Instead, we use the general syntax for non-uniform types, and we assume the existence of predicates on types $client(T\zeta)$ and $server(T\zeta)$ that return as result whether the type is a client or a server type. In practice, defining such predicates is easy, since public types are of the form: $M()\zeta$ for any ζ, with named type M recursively defined as $M() \triangleq ([\top, \bot]\pi_{\updownarrow}.\, M())$. We also define $public(T\zeta) \triangleq client(T\zeta) \vee server(T\zeta)$, and $private(T\zeta) \triangleq \neg public(T\zeta)$.

5.4.2 Type Compatibility and Context Composition

Devising a sound typing rule for parallel composition implies two independent tasks. The first and simpler one is to check than in the system each name is present only in an authorized multiplicity: one server and potentially many server for public ports, and at most two occurrences for a private port. The second task is to check that private ports have only occurrences with compatible types.

We start by defining type compatibility, which is the most difficult part of the system. This relation is hardly definable in its semantic form, and we prefer to characterize it axiomatically. For the sake of simplicity, in the given rules we leave implicit the syntactic mechanism necessary to deal with recursion; such mechanism is assumed to be provided by a context identical to the context E used in Section 5.3 to axiomatize subtyping. The essential rules in our system are the ones for dealing with prefixes. What is checked is very simple: whenever a type can let time pass while exhibiting a requirement on its environment, a compatible type should offer a matching offer before the requirement reaches its deadline (if any).

The judgments of our axiom system adopt the form: $Comp_{\tilde{M}\#\tilde{N}}(T\zeta, U\xi)$ where $\tilde{M} \triangleq M_1, \ldots, M_k$ and $\tilde{N} \triangleq N_1, \ldots, N_l$ are lists of named types. Each list is a context representing the stack of calls that have been done in the past since the last discrete action took place. The left-most list $\tilde{M}$ is the list for $T\zeta$, and $\tilde{N}$ is the list for $U\xi$. The motivation of maintaining such lists is to check that each constraint with an unbounded urgency condition, although it does not impose a firm deadline on the occurrence of the action, gets satisfied eventually. The prefixes with unbounded urgency condition can be seen as fairness constraints that are imposed to the execution. Our compatibility axiom system hence checks that a recursive call can not be done if an unboundedly urgent action has not found a matching offer from the peer type. If empty, we allow to simply forget the context, writing $Comp(T\zeta, U\xi)$.

Definition 5.4.1. *The possibility of inferring $Comp(T\zeta, U\xi)$ by using the rules of Table 5.4 is what defines* type compatibility.

Please notice again that in the rules of Table 5.4, we leave implicit the treatment of fix-point computations: a list of recursive calls to $Comp$ is actually maintained, that allows to discharge an antecedent to a rule whenever a deduction tree produced it as a consequent lower in the proof tree. As an example, when proving $Comp(T\zeta, U\xi)$, we make this judgment appear at the bottom of the proof tree, consequent of the lower proof rule. If, up in the tree, we need to show it as an antecedent

$$\text{Prefix}\ \frac{\begin{array}{c}TimeComp(\zeta,\sigma,\upsilon,\pi_\updownarrow,\xi,\sigma',\upsilon',\pi'_\updownarrow)\\ Next(([\sigma,\upsilon]\pi_\updownarrow.T),\zeta,([\sigma',\upsilon']\pi'_\updownarrow.U),\xi,\tilde{M},\tilde{N})\end{array}}{Comp_{\tilde{M}\#\tilde{N}}(([\sigma,\upsilon]\pi_\updownarrow.T)\zeta,([\sigma',\upsilon']\pi'_\updownarrow.U)\xi)}$$

$$\text{Reset}\ \frac{Comp_{\tilde{M}\#\tilde{N}}(T\,\zeta^{\downarrow x}=U\,\xi^{\downarrow x})}{Comp_{\tilde{M}\#\tilde{N}}((\nu x\ T)\zeta,((\nu x\ U)\xi))}$$

$$\text{Rec-R}\ \frac{\begin{array}{c}((\upsilon \not\Leftrightarrow \bot)\wedge(\upsilon^{\uparrow}\Leftrightarrow\upsilon))\Rightarrow N\notin\tilde{N}\\ Comp_{\tilde{M}\#cons(N,\tilde{N})}(([\sigma,\upsilon]\pi_\updownarrow.T)\zeta,(U[\tilde{x}/\tilde{y}])\xi^{/\tilde{x}})\end{array}}{Comp_{\tilde{M}\#\tilde{N}}(([\sigma,\upsilon]\pi_\updownarrow.T)\zeta,(N(\tilde{x})\xi))}\ N(\tilde{y})\triangleq U$$

$$\text{Part-}\zeta\text{-R}\ \frac{Comp_{\tilde{M}\#\tilde{N}}(T\zeta,U\xi)\quad Comp_{\tilde{M}\#\tilde{N}}(T\zeta,U\xi')}{Comp_{\tilde{M}\#\tilde{N}}(T\zeta,U(\xi\vee\xi'))}$$

$$\text{Absurd-}\zeta\ \frac{-}{Comp_{\tilde{M}\#\tilde{N}}(T\bot,U\bot)}\qquad \text{Relax}\ \frac{Comp_{\tilde{M}\#\tilde{N}}(T\zeta,U(\xi\vee\xi'))}{Comp(T\zeta,U(\xi\vee\xi'))}$$

$$\Box\triangleq\left\{\begin{array}{l}x\notin(clocks(\zeta)\cup clocks(\xi))\ \text{and}\\ (((\zeta\Rightarrow\upsilon)\vee(\xi\Rightarrow\upsilon'))\Rightarrow((\zeta'^{\backslash x}\not\Leftrightarrow\zeta)\wedge(\xi'^{\backslash x}\not\Leftrightarrow\xi)))\end{array}\right.$$

Table 5.4: Axiomatic Definition of Type Compatibility

to a rule, it means that we reached a fix-point, and we can discharge that antecedent immediately. This comes from standard results on the existence of unique fix-point solution for well-guarded processes [Mil89a].

We now give an informal explanation of each rule. The first rule deals with prefix actions, in a similar way to the prefix rule for subtyping. The goal here is however to decide whether the requirements of the first parameter to $Comp$ receives matching offers in time from the second parameter. The rule makes use of predicates $TimeComp$ and $Next$, that we define below:

$$\begin{array}{l}TimeComp(\zeta,\sigma,\upsilon,\pi_\updownarrow,\xi,\sigma',\upsilon',\pi'_\updownarrow)\triangleq\\ \left(\bigvee_{R\in Ttype}(\text{ante}(R))(\zeta,\sigma,\upsilon)\right)\wedge\left(\bigvee_{R\in Ttype}(\text{ante}(R))(\xi,\sigma',\upsilon')\right)\wedge\\ ((\zeta^{\uparrow}\wedge\upsilon\not\Leftrightarrow\bot)\Rightarrow(Comp(\pi_\updownarrow,\pi'_\updownarrow)\Rightarrow((((\xi^{\downarrow x})^{\uparrow}\wedge\sigma')^{/x}\wedge((\zeta^{\downarrow x})^{\uparrow}\wedge\upsilon)^{/x})\not\Leftrightarrow\bot))\wedge\\ \qquad\qquad(\neg Comp(\pi_\updownarrow,\pi'_\updownarrow)\Rightarrow((((\zeta^{\downarrow x})^{\uparrow}\wedge\upsilon)^{\uparrow}\wedge((\xi^{\downarrow x})^{\uparrow}\wedge\sigma'))\Leftrightarrow\bot)))\wedge\\ ((\xi^{\uparrow}\wedge\upsilon'\not\Leftrightarrow\bot)\Rightarrow(Comp(\pi_\updownarrow,\pi'_\updownarrow)\Rightarrow((((\xi^{\downarrow x})^{\uparrow}\wedge\upsilon')^{/x}\wedge((\zeta^{\downarrow x})^{\uparrow}\wedge\sigma)^{/x})\not\Leftrightarrow\bot))\wedge\\ \qquad\qquad(\neg Comp(\pi_\updownarrow,\pi'_\updownarrow)\Rightarrow((((\xi^{\downarrow x})^{\uparrow}\wedge\upsilon')^{\uparrow}\wedge((\zeta^{\downarrow x})^{\uparrow}\wedge\sigma))\Leftrightarrow\bot)))\end{array}$$

where the predicate $Comp(\pi_\updownarrow,\pi'_\updownarrow)$ is defined by

$$Comp(\pi_\updownarrow,\pi'_\updownarrow)\triangleq\left\{\begin{array}{l}(\pi_\updownarrow=\uparrow\mathcal{T})\Rightarrow(\pi'_\updownarrow=\downarrow\mathcal{T}'\wedge\mathcal{T}\sim\mathcal{T}')\wedge\\ (\pi_\updownarrow=\downarrow\mathcal{T})\Rightarrow(\pi'_\updownarrow=\uparrow\mathcal{T}'\wedge\mathcal{T}\sim\mathcal{T}')\end{array}\right.$$

This effectively allows to check that any requirement from one process with find an overlapping offer from its peer type. The next predicate ensures that both types are still compatible under any type evolution. The discrete actions are performed indepentently, while time-passing actions must go at the same pace.

$$Next(T,\zeta,U,\xi,\tilde{M},\tilde{N}) \triangleq$$
$$((T\zeta^{\downarrow z} \xrightarrow{\langle\pi_\updownarrow\rangle} T'\zeta' \wedge U\xi^{\downarrow z} \xrightarrow{\langle\pi'_\updownarrow\rangle} U'\xi' \wedge (\zeta'^{/z} \Leftrightarrow \xi'^{/z})) \Rightarrow (Comp_{\tilde{M}\#\tilde{N}}(T'\zeta'^{\setminus z}, U'\xi'^{\setminus z})) \wedge$$
$$(T\zeta \xrightarrow{\pi_\updownarrow} T'\zeta' \Rightarrow Comp(T'\zeta', U\xi)) \ \wedge (U\xi \xrightarrow{\pi'_\updownarrow} U'\xi' \Rightarrow Comp(T\zeta, U'\xi'))$$

The rule Reset allows types to set a clock to the null value at the same time. The rule is clearly sound, but it is also complete because a type may always reset a clock at each step: it can simply ignore it later on if it does not need it. The rule Rec deals with named type invocation and recursion. An invocation is correct if the result of this invocation has already been proved, *i.e.* , we have reached a fix-point. The condition on the rule ensures that a prefix with an unbounded urgency condition may not persist along a recursive behavior that will never satisfy it. We have not represented rule Rec-L in Table 5.4, which is the rule similar to Rec-R that allows to perform recursion on the left-most parameter to *Comp*.

The rule Part-ζ-R is an ancillary rule allowing to refine the state-space if necessary. As for the recusrion rule, we did not represent its sibling Part-ζ-L. The rule Absurd-ζ makes any two types compatible if there are no possible clock values that satisfy their clock contexts. Finally, the rule Relax allows to get rid of the stack of named type calls, if necessary. From the above rules, one can easily see that type compatibility is preserved under type reduction.

We now define type composition $\Gamma_1 \oplus \Gamma_2$ as the point-wise union of Γ_1 and Γ_2. This union is defined under the conditions of good multiplicity exposed earlier. Hence, depending on the fact that for some name a, $\Gamma_1(a) = T_1\,\zeta_1$ and $\Gamma_2(a) = T_2\,\zeta_2$, $\Gamma_1 \oplus \Gamma_2$ is defined when:

- if $server(T_1\zeta_1)$ then $client(T_2\zeta_2)$, and vice-versa;
- if $private(T_1\zeta_1)$ when and only when $private(T_2\zeta_2)$;
- if two ports are private, then their types are compatible, $Comp(T_1\zeta_1, T_2\zeta_2)$.

In the case of public ports, only the server role is put in the resulting typing environment:

$$(\Gamma_1, a : T_1\zeta_1) \oplus (\Gamma_2, a : T_2\zeta_2) \triangleq (\Gamma_1 \oplus \Gamma_2, a : T_1\zeta_1) \text{ if } public(\Gamma_1\zeta_1) \text{ and } client(T_2\zeta_2)$$

In the case of compatible private ports, the result is $a : *$, under the condition that the types are compatible:

$$(\Gamma_1, a : T_1\zeta_1) \oplus (\Gamma_2, a : T_2\zeta_2) \triangleq (\Gamma_1 \oplus \Gamma_2, a : *) \text{ if } Comp(\Gamma_1\zeta_1, T_2\zeta_2)$$

The latter condition ensures that interacting well-typed processes shall be well-behaved. Having an uncomposable type as a result ensures that one may will never be able to introduce another copy of name a in the system. Consequently, private types yield only pairwise associations.

5.4.3 The Type System

The main purpose of our type system is to detect and to forbid behaviors where a requirement is failed to be satisfy in time because processes do not synchronize enough. The result of such a misconduct is the intervention of an *Error* process, deadlocking the specification. In order to guarantee the total absence of such deadlocks, we forbid the *Error* process to occur on its own in a term, following an action prefix; no process containing *Error* is well-typed. Detecting all errors without that condition would be unfeasible, since our processes yield infinite state space and are hence out-of-scope for reachability analysis methods.

Essentially, the offers of a type must be included in the offers of a corresponding well-typed process, while the requirements of a process must be included in the requirements of the corresponding type. The employed mechanism is therefore similar to the one used in the axiomatization of subtyping.

However, in order enforce the inclusion rule for offers upon processes that employ the choice operator, we need to devise a purely syntactical way of annotating processes to indicate at any step which offers have been satisfied. We choose to underline private port types which offer is not satisfied by one sub-process in a choice. We write $\underline{\Gamma}$ if and only if all private ports in Γ make offers that are not satisfied. The set of possible actions of a typing context and the result they yield are not modified by underlining: if $\Gamma, a : T\zeta \xrightarrow{\lambda_\Gamma} \Gamma', a : T'\zeta'$, then $\Gamma, \underline{a : T\zeta} \xrightarrow{\lambda_\Gamma} \Gamma', \underline{a : T'\zeta'}$.

This gives us the only axiom of the type system:

$$Ax \quad \frac{-}{\underline{\Gamma} \vdash 0\,\zeta}$$

This axiom says that the idle process 0 never constrains its environment, and is therefore compliant with any type on this point; but the idle process also never satisfies any requirement, by never offering the actions that it should as specified by Γ. Hence, all private port types in Γ must be underlined. We shall write that a process $t\,\zeta$ is *well-typed* under context Γ if and only if $\Gamma \vdash T\zeta$ and no port type in Γ is underlined. Hence, 0 can be well-typed only if it can be proved that all the port types in Γ can make no offer but only force interaction.

We now present the rules dealing with inputs and outputs. The idea behind them is similar to the Comm rule of Table 5.3. However, here we have to range over two symbolic semantics: the symbolic semantics for types (Table 5.2), and the symbolic semantics for processes (essentially, Table 4.3). Hence, the predicate we use to check timeliness constraints refers to rules of the set $Tpref \triangleq \{$TSEL, TURG, TELA, TIDL, TUNBS, TUNBU, TMISS, $\text{TREQ}_{max}\}$ represented in Tables 4.3 and 4.4, as well as rules of the set $Ttype \triangleq \{$TSEL, TURG, TELA, TSKIP, TUNBS, TUNBU, $\text{TREQ}_{max}\}$ given in Table 5.2. Tough we have (intentionally) given the same name to different rules, it should always be clear from context which is meant.

Before we give the rules, we need some ancillary definitions. Again we take source and destination zones as written in each rule R of $Ttype$ or $Tpref$ to be the functions ante(R) and dest(R) of the constraints $(\zeta, \sigma, \upsilon)$ appearing in them. For some x not in $clocks(\zeta) \cup clocks(\xi)$, we define the predicate Conform, that check the

timeliness of offers and requirements during time progress, by:

$$\begin{array}{l}
\mathrm{Conform}(\zeta,\sigma,\upsilon,\pi_{\updownarrow},\xi,\sigma',\upsilon',\pi'_{\updownarrow}) \triangleq \\
\left(\bigvee_{R\in Tpref}(\mathrm{ante}(R))(\zeta,\sigma,\upsilon)\right) \wedge \left(\bigvee_{R\in Ttype}(\mathrm{ante}(R))(\xi,\sigma',\upsilon')\right) \wedge \\
((\xi^{\uparrow}\wedge\sigma' \not\Leftrightarrow \bot) \Rightarrow ((\pi_{\updownarrow}\preceq\pi'_{\updownarrow}) \Rightarrow (((\zeta^{\uparrow}\wedge\sigma) \not\Leftrightarrow \bot) \wedge ((\xi^{\downarrow x})^{\uparrow}\wedge\sigma')^{/x} \Rightarrow ((\zeta^{\downarrow x})^{\uparrow}\wedge\sigma)^{/x})) \wedge \\
\qquad ((\pi_{\updownarrow}\not\preceq\pi'_{\updownarrow}) \Rightarrow ((((\xi^{\downarrow x})^{\uparrow}\wedge\sigma') \wedge ((\zeta^{\downarrow x})^{\uparrow}\wedge\sigma)^{\uparrow}) \Leftrightarrow \bot))) \wedge \\
((\zeta^{\uparrow}\wedge\upsilon \not\Leftrightarrow \bot) \Rightarrow ((\pi_{\updownarrow}\preceq\pi'_{\updownarrow}) \Rightarrow (((\xi^{\uparrow}\wedge\upsilon') \not\Leftrightarrow \bot) \wedge ((\xi^{\downarrow x})^{\uparrow}\wedge\upsilon')^{/x} \Rightarrow ((\zeta^{\downarrow x})^{\uparrow}\wedge\upsilon)^{/x})) \wedge \\
\qquad ((\pi_{\updownarrow}\not\preceq\pi'_{\updownarrow}) \Rightarrow ((((\zeta^{\downarrow x})^{\uparrow}\wedge\sigma) \wedge ((\xi^{\downarrow x})^{\uparrow}\wedge\sigma')^{\uparrow}) \Leftrightarrow \bot)))
\end{array}$$

The following predicate $Next$ imposes that, whatever the discrete action of a type or a process can be, the resulting type and process continuations must still be bound by the "well-typed" relation. The predicate following $\forall\xi'_i \in \Gamma' \ldots$ in $Next$ is taken to be true for all ξ'_i such that there exists a'_i and T'_i with $\Gamma'(a'_i) = T'_i\zeta'_i$. Therefore, $Next$ is defined by:

$$\begin{array}{l}
Next(\Gamma,t,\zeta,n_{new},\mathcal{T}_{new},U,\xi) \triangleq \\
\quad (\Gamma^{\downarrow z} \xrightarrow{\mathcal{S}_\Gamma} \Gamma' \Leftrightarrow (\exists t'\,\zeta'.\ t\,\zeta^{\downarrow z} \xrightarrow{\mathcal{S}} t'\,\zeta' \wedge (\forall\xi'_i\in\Gamma'.\ (\xi'^{/z}_i \Leftrightarrow \zeta'^{/z})))) \wedge \\
\quad (\Gamma^{\downarrow z} \xrightarrow{\mathcal{S}_\Gamma} \Gamma' \Rightarrow (\Gamma'^{\backslash z} \vdash t'\,\zeta'^{\backslash z})) \wedge \\
\quad ((t\,\zeta \xrightarrow{\pi,\mu} t'\,\zeta') \Rightarrow (new(\Gamma,\pi), n_{new} : \mathcal{T}_{new} \vdash t'\,\zeta')) \ \wedge ((\Gamma \xrightarrow{n\pi_{\updownarrow}} \Gamma') \Rightarrow (\Gamma' \vdash t\,\zeta))
\end{array}$$

where we have to define $new(\Gamma,\pi)$. The goal of this function is to modify Γ so that the constraints on port multiplicity in a well-typed configuration be respected. That is, sent private ports or server ports are lost by the sending process, while client ports are duplicated. There is a condition of well-definedness of $new(\Gamma,\pi)$ that the type of a passed argument must be a subtype of the corresponding formal parameter: if we take $subj(\pi) = a$, $obj(\pi) = b$, $\Gamma(a) = T\zeta$, with $T \triangleq [\sigma,\upsilon]\pi_{\updownarrow}.M(\tilde{x})$, then if $\Gamma(b) \equiv U\xi$ we must have $\uparrow U\xi \preceq \pi_{\updownarrow}$ or $\downarrow U\xi \preceq \pi_{\updownarrow}$, and similarly for $obj(\pi) = n$. Under those conditions:

$$new(\Gamma,\pi) \triangleq \begin{cases}
\Gamma\backslash\{b : U\xi\} \textbf{ if } (\pi \equiv \overline{a}b \wedge \Gamma(b) \equiv U\xi \wedge (private(U\xi) \vee server(U\xi))) \\
\Gamma \textbf{ if } ((\pi \equiv \overline{a}n \wedge \Gamma(n) \equiv \mathcal{T} \wedge (client(\mathcal{T}) \vee clock(\mathcal{T}))) \vee (\pi \equiv \overline{a}(b))) \\
\Gamma, b : U\xi \textbf{ if } (\pi \equiv a(b) \wedge b \notin dom(\Gamma))
\end{cases}$$

We need two different rules for output, one for free output dealing with arguments annotated with public types, and the other for bound output dealing with arguments annotated with private types. The bound output sends a private port of type $U_1()$, and creates another private port with the same name b but with type U_2, that is kept locally.

$$Boutput\ \frac{\begin{array}{l} U \triangleq [\sigma',\upsilon']\pi'_{\updownarrow}.M(\tilde{x}) \\ \mathrm{Conform}(\zeta,\sigma,\upsilon,\uparrow U_1(),\xi,\sigma',\upsilon',\pi'_{\updownarrow}) \\ Next((\Gamma,a : U\xi),([\sigma,\upsilon]\overline{a}(b^{U_1,U_2}).\,t),\zeta,b,U_2(),U,\xi) \end{array}}{\underline{\Gamma}, a : U\xi \vdash ([\sigma,\upsilon]\overline{a}(b^{U_1,U_2}).\,t)\zeta}$$

The free output may only apply to clocks or public ports, either clients or servers. If a client is transmitted, its type is kept locally, and the name can still be used (a

copy is done). If a server is sent away, then the capacity to use it is lost, since for a given port name there must be at most one server in any well-typed configuration. We use the empty-set symbol in place of a newly created name and its type in the invocation of *Next*; this is because no new name is created.

$$Foutput\ \frac{\begin{array}{l} U \triangleq [\sigma', \upsilon']\pi'_{\updownarrow}.M(\tilde{x}) \\ \mathrm{Conform}(\zeta, \sigma, \upsilon, \uparrow \Gamma(b), \xi, \sigma', \upsilon', \pi'_{\updownarrow}) \\ Next((\Gamma, a : U\xi), ([\sigma, \upsilon]\overline{a}n.\, t), \zeta, \emptyset, \emptyset, U, \xi) \end{array}}{\underline{\Gamma}, a : U\xi \vdash ([\sigma, \upsilon]\overline{a}n.\, t)\zeta}$$

The input rule takes into account the indicated type for the received name. It is otherwise written on the same model as the two output rules.

$$Input\ \frac{\begin{array}{l} U \triangleq [\sigma', \upsilon']\pi'_{\updownarrow}.M(\tilde{x}) \\ \mathrm{Conform}(\zeta, \sigma, \upsilon, \downarrow \mathcal{T}, \xi, \sigma', \upsilon', \pi'_{\updownarrow}) \\ Next((\Gamma, a : U\xi), ([\sigma, \upsilon]a(n^{\mathcal{T}}).\, t), \zeta, \emptyset, \emptyset, U, \xi) \end{array}}{\underline{\Gamma}, a : U\xi \vdash ([\sigma, \upsilon]a(n^{\mathcal{T}}).\, t)\zeta}$$

The name introduction rules action either upon clock names or upon public port names. For a port name, the left-most rule simply extends the typing context with the new name, which must be public. The other rule resets the clock in the current clock context ζ of the process.

$$NewPort\ \frac{\begin{array}{l} public(T()) \\ \Gamma, a : T() \vdash t\,\zeta \end{array}}{\Gamma \vdash (\nu a^T\ t)\zeta} \qquad NewClock\ \frac{\Gamma \vdash t\,\zeta^{\downarrow x}}{\Gamma \vdash (\nu x\ t)\zeta}$$

Recursive calls are typed using the unique fix-point property, which is valid since our processes exhibit only *well-guarded recursion* (see Section 3.2). First, the rule on the left states that a named process is well-typed under some environment Γ if its term is well-typed under Γ augmented with the names and types of the parameters received by the process upon invocation. The right-most rule states that if a named process is well-typed under the current environment and if the names that are passed to it upon invocation are subtypes of the formal parameters specified by the process, then the invocation is well-typed.

$$Named\ \frac{\Gamma, A(\tilde{\mathcal{T}}), \tilde{m} : \tilde{\mathcal{T}} \vdash t}{\Gamma \vdash A(\tilde{\mathcal{T}})}\ A(\tilde{m} : \tilde{\mathcal{T}}) \triangleq t \qquad Id\ \frac{\begin{array}{l} \Gamma \vdash A(\tilde{\mathcal{T}}) \\ \Gamma \vdash \tilde{n} : \tilde{\mathcal{T}}' \end{array}}{\Gamma \vdash A(\tilde{n})\zeta}$$

The choice can perform non-deterministic computations, but the continuation of each branch t_1, t_2 must respect the same protocol, the one that is given by a sort of union sum of their typing contexts: $\Gamma_1 \uplus \Gamma_2$. This union is defined whenever the two contexts Γ_1 and Γ_2 have the same domain. For any a included in the domain of Γ_1 and Γ_2, if $\underline{a : \mathcal{T}} \in \Gamma_1$ and $\underline{a : \mathcal{T}} \in \Gamma_2$, then $\underline{a : \mathcal{T}} \in \Gamma_1 \uplus \Gamma_2$, and otherwise $a : \mathcal{T} \in \Gamma_1 \uplus \Gamma_2$. The rule therefore allows different offers to be satisfied by different

branches in the choice. Intuitively this is sufficient, since it is only needed that one branch make an offer on a to actually satisfy a type requiring an offer on a.

$$Sum\ \frac{\begin{array}{c}\Gamma_1 \vdash t_1\,\zeta \\ \Gamma_2 \vdash t_2\,\xi \\ (\zeta \wedge \xi) \not\Leftrightarrow \bot\end{array}}{\Gamma_1 \uplus \Gamma_2 \vdash (t_1 + t_2)(\zeta \wedge \xi)}$$

Parallel composition ensures that port names appear in the system with multiplicities in compliance with the rules given by their types. This parallel composition rule is similar to a logical "cut" rule: when successfully applied, a cut is made on private names that have found a compatible peer role in the other parallel component. Their type then becomes $*$, making them henceforth uncomposable.

$$Par\ \frac{\begin{array}{c}\Gamma_1 \vdash t\,\zeta \\ \Gamma_2 \vdash u\,\xi \\ (\zeta \wedge \xi) \not\Leftrightarrow \bot \\ \Gamma_1 \oplus \Gamma_2 \text{ defined}\end{array}}{\Gamma_1 \oplus \Gamma_2 \vdash (t \mid u)(\zeta \wedge \xi)}$$

As we will see in the next section, this rule for parallel composition is very important since it allows to perform *compositional* reasoning over processes. Having such a way of reasoning allows to divide a given specification into independent modules and prove them independently one from the other, but also to study partial specifications and have a way of deciding whether they can be composed safely or not. In our case, the cornerstone of composition and decomposition theorems is the obligation of compatibility between private ports that bear the same name and occur in different components.

Finally, we add ancillary rules that allow to perform subtyping and to refine the state space if it is too coarse.

$$Subtype\ \frac{\begin{array}{c}\Gamma' \vdash t\,\zeta \\ \Gamma' \preceq \Gamma\end{array}}{\Gamma \vdash t\,\zeta} \qquad Refine\ \frac{\Gamma \vdash t\,\zeta}{\Gamma \vdash t\,\zeta'}$$

5.5 Properties of our Type System

We start by the well-known *subject reduction* property first stated by Curry.

Theorem 5.5.1 (Subject Reduction). *if* $\Gamma \vdash t\,\zeta$*, then* $t\zeta \xrightarrow{\lambda} t'\zeta'$ *implies that there exists* Γ' *such that* $\Gamma' \vdash t'\,\zeta'$*.*

The proof of the subject reduction theorem goes as usual by induction on the derivation of $t\zeta \xrightarrow{\lambda} t'\zeta'$, with a case analysis on the last rule used. The most difficult cases are the choice operator and the parallel composition operator. For the choice operator it involves showing that, after composition, the only unsatisfied port types are the underlined one. For the parallel rule, we have to show that subject reduction preserves the well-formedness of the typing context: the addition of contexts and the

input/output rules must ensure only safe duplication and access to names. This is the case since the input/output rules extend the environment with the proper type whenever a port is created, and forbids later access to sent private or server port. This appears in the function $new(\Gamma, \pi)$ used by predicate $Next$. Then the $\oplus$ context addition imposes the uniqueness of server and private names in parallel components.

The subject reduction property therefore only takes into account (and reflects) a small part of the relations between types and processes. We can also prove the less usual "type reduction" property: since our types can evolves separately from our processes, we must be able to prove that well-typing is preserved by type reduction.

Theorem 5.5.2 (Type reduction). *if* $\Gamma \vdash t\,\zeta$, *then* $\Gamma \xrightarrow{a\pi_\updownarrow} \Gamma'$ *for some* a *and* $\pi_\updownarrow$ *implies* $\Gamma' \vdash t\,\zeta$.

The subject and type reduction property allows to prove the typical safety property stating that a closed configuration may not produce errors. This property is a safety property because it says that, *whenever the process is willing to take a transition*, then its type may evolve (or stay unchanged) so that the resulting process is well-typed.

Theorem 5.5.3 (Run-Time Safety). $a_1 : *, \ldots, a_k : *, A_1(\mathcal{T}_1), \ldots, A_l(\mathcal{T}_l) \vdash P$ *implies* $P \not\longrightarrow^* Error$.

The proof goes by induction, showing that if at the current step P is well typed, then it can not produce an error, and it will be well-typed after any action it can produce. The last part is provided by the subject reduction theorem. Proving that a well-typed process can not perform an action leading to an error in the next state can be done by induction on the structure of terms, reasoning about the transitions allowed by their symbolic semantics.

The essential rule in that context is the parallel rule for typing. It allows to decompose the proof by traditional assume/guarantee reasoning for safety properties [MC81]. Assume that two processes P_1 and P_2 have in common a private port a and that $P \triangleq P_1 \mid P_2$ is well-typed under an environment $\Gamma, a : *$. Type of a in Γ is mandatorily $*$, since a has a private type, and it appears in both P_1 and P_2. From the parallel typing rule, we must find $T_1\zeta_1$ and $T_2\zeta_2$ so that $\Gamma_1, a : T_1\zeta_1 \vdash P_1$ and $\Gamma_2, a : T_2\zeta_2 \vdash P_2$ and the round sum of the contexts be equal to Γ. To simplify, we will assume the existence of Γ_1 and Γ_2.

The compatibility of types $T_1\zeta_1$ and $T_2\zeta_2$ then gives us a sufficient condition for performing assume/guarantee reasoning on P_1 and P_2. Assume that P_1 and P_2 are well typed. In the current state, type compatibility ensures that neither P_1 nor P_2 can perform an error, since the requirements of $T_1\zeta_1$ are always overlapping with matching offers from $T_2\zeta_2$, and vice-versa. By subject reduction and type reduction, we can show that any configuration $\Gamma_1', a : T_1'\zeta_1'$ to which the context can evolve in one step and any P_1' to which P_1 can evolve in one step are so that $\Gamma_1', a : T_1'\zeta_1' \vdash P_1$ and $\Gamma_1, a : T_1\zeta_1 \vdash P_1'$. Hence in the next step, P_1 will always be well-typed. By performing similar reasoning on P_2 and by preservation of the type compatibility under type reduction, we obtain the sought result through an ad-hoc mutual induction.

If this way of reasoning is sound, it is certainly not complete. This conjecture comes from recent results [NT00] that show the very general rule devised by McMillan [McM99] as being incomplete.

Finally, we shortly address type-checking decidability. To prove that type-checking terminates, we have to prove that typing contexts do not grow unboundedly with recursive calls. We obtain this result by observing two facts. The first is that the state-space of any type is *finite*: there are, given any set of clocks, only finitely-many possible zones, depending on the ceiling value for each clock appearing in the type. At any moment, the set of clocks referred to by a type is finite, and it has a maximum because the size of a type term can not grow unboundedly. It is sufficient to reference the constraints on the current clock names of the type in its clock constraint environment. This is also true for named process invocation: when a recursive process name invocation is done, only a finite number of existing names are passed to the resulting continuation, and all the others can be forgotten.

5.6 Liveness and Composition

In Chapter 3 we introduced operational semantic rules that yield timed transition systems. A transition system only predicts the *possible* actions of the corresponding process, stating only *safety* properties for it. This fact is common to all the process algebras that we know, where *liveness* properties are purely ignored (except [Par85]). However, liveness properties are of prime importance, especially when dense time is involved. To study liveness properties on our models, we need to perform two technical adjustments:

- in addition to unbounded, finite executions, we have to explicitly consider *infinite* executions,
- we retain a *weak fairness* assumption [LPS81] on silent actions when they are urgent.

The weak fairness assumption states that no infinite sequence of transitions can be performed with an urgent action enabled without this action being performed. This forces the system to progress whenever an urgent action should be performed; it has for result either the occurrence of the urgent action, or the occurrence of an error. Without this assumption, zeno executions are produced because time can get closer and closer to the upper-bound of the urgency condition without actually reaching it.

Using this assumption, we can prove the absence of deadlocks in our specifications. Indeed, type-checking ensures that a process can always progress to satisfy the requirements of its environment if all its own requirements are satisfied. Processes have two immediate ways to prevent time from diverging: they can produce an error, or they can perform a silent action with an urgency condition. We tackle the first through type-checking, while the second is prevented by the fairness assumption.

To ensure that time always diverges, we have to make one more assumption. If we impose processes to be *strongly non-zeno* [HNSY92, SY96], then we can extend our assume/guarantee reasoning to the proof that a composition is non-zeno from

the non-zenoness of its components. Strong non-zenoness is a convenient syntactic criterion to establish the non-zenoness of a regular process t: it consists in checking that any recursive call in t intervenes after at least one time unit has elapsed. This can be obtained by resetting a clock x in t and then allowing to leave some continuation of y only if x is superior to 1. The reason for non-zenoness obligation is the possibility for processes to immoderately interact on public ports, which can also prevent time from diverging.

Theorem 5.6.1 (Non-zenoness of configurations). *Take the rules of the type systam as given, but restrict the composition $\oplus$ of processes to strongly non-zeno types. We obtain that when $a_1 : *, \ldots, a_k : *, A_1(\mathcal{T}_1), \ldots, A_l(\mathcal{T}_l) \vdash t\,\zeta$ and all $\mathcal{T}_1, \cdots, \mathcal{T}_l$ are strongly non-zeno, $t\,\zeta$ is non-zeno.*

(Sketch of the proof). The proof of the theorem is similar to the proof of the run-time saftey theorem of the previous section. The difficult part still is the rule for parallel composition. We again employ assume/guarantee reasoning. Strong non-zenoness is a syntactic property of our type terms, that is preserved by parallel composition and assembly. We then have to prove that any evolution from one process $t_1\,\zeta$ may not indefinitely block time progression of some process $t_2\,\zeta$ in $(t_1 \mid t_2)\zeta$, and the converse proposition. The proof thus proceeds by reasoning on the transitions that can be inferred for their parallel composition $(t_1 \mid t_2)$ from the respective transitions of t_1 and t_2. Relying on the good-typing assumption, subject reduction, and the fairness assumption on silent transitions with urgency, we can show that either t_1 may let time pass so that the value of one clock progresses of one time unit, or it may not let time pass because of an urgency condition on some free name, and then the other process in parallel has a matching offer. This can be repeated only a finite number of times, since the type of all the port names appearing in t_1 are strongly non-zeno: t_1 must "follow" a type whenever this type makes an offer, and thus it must let time pass forever (an offer prefix may not block time progression), and it has meanwhile to exhibit less prefixes with an enabled urgency condition, that may block time. Thus t_1 has to provide fewer time-blocking instructions than its type, meaning that if one provides it a closed environment, it is also non-zeno. Consequently, t_1 may forbid t_2 to let time pass only finitely-many times. By applying the same reasoning to t_2, and by mutual induction, we obtain the sought result. □

From the rules of the type system, it can be seen easily that the predicate Conform indeed ensures the correct behavior of the typed process in the future whenever this process may let time pass. Such assume/guarantee reasoning can be seen as an application of McMillan's rule [McM99], where the assumed properties at state n are the strong-zenoness and the good typing, that allow to prove time progression at state $n + 1$. The only change is therefore that we apply this rule in a timed setting, proving that some process will always agree to let one unit of time pass after finitely-many discrete transitions if it is well-typed. The possibility of performing a discreticized time-passing transition hence intervenes only in the guarantee part of the reasoning at each step.

5.7 Related Works and Conclusion

To provide a complete reference over the works in the field we be nearly impossible. We just mention a few results about compositional and assume/guarantee reasoning, and on (behavioral) type systems for mobile processes.

The formal definition of behavioral typing originates from Nierstrasz [Nie95]. The general principle is to consider types as specifications, or abstractions of the typed behaviors. Hence processes can not only denote behaviors where certain actions are always enabled or always prohibited, but they can take into account the current context to determine if performing an operation is legal or not. This is what is called a *non-uniform* service provision [Nie95].

The community mainly stems from people studying the principles of programming languages, especially object-oriented programming. Indeed, when encoding objects of λ-calculus terms into the π-calculus, many people were faced with the impossibility of processes to control the actions they make available to their environment. In this respect, behavioral types can then be viewed as a deflection from *type-and-effect* systems [TJ94]. The goal of the abstraction is then to ensure that the environment of a process will not block an action *ad-infinitum*. Modal types can also be found as support for compositional bisimulation proof systems of processes with infinite state-spaces: we can cite Micculan and Gadducci [MG95], as well as Mads Dam [Dam95], and Dam and Amadio [AD96]. However, type-checking is then undecidable, since deciding bisimilarity is unfeasible.

A related source of research in the programming languages community was the *actor* model [Hew77, Agh86, AMST97]. Actors are autonomous entities that each possess a FIFO message channel identified by a unique address. When executing, an actor may examine its channel, take a message from it, and continue by sending messages to other actors and then behaving as ordered by its continuation. The sending of a message to an actor can then be perceived as an obligation put by the sender on the receiver to eventually accept the sent message.

Many solutions have been proposed in the last few years, for both object-oriented and actor-oriented formalisms. They converge in the fact that they use essentially the π-calculus or one of its variants. Original type systems for the π-calculusonly focus on restraining the capacity of channels to send, or receive, or both [PS93]. Some works then try to check that processes are *input-enabled*, that is they are always able to receive the messages they can be addressed; their types do not denote modal properties of processes. We can cite among them Sangiorgi [San99], and Boudol *et al.* [BAL99]. Extensions of similar types to access control of resources with security-based types have been also proposed [YH00, HR02, TZH02, BC01]. A recent survey on non-behavioral types for the π-calculus can be found in [Gay99].

5.7.1 Behavioral Types

In the behavioral typing community, works can be divided between the ones that type processes and the ones that type ports. Port types are used essentially in actor-based formalisms, that allow only asynchronous communication. The regular types of Nierstrasz [Nie95] belong to this kind. Since then, Ravara and Vasconcelos

have proposed solution for non-uniform service interactions in the language TyCO [VB98, RRV99, RV97, RV00], and more recently with Simon Gay on "session types" [GVR02, VVR02]. The works of Ravara and Vasconcelos use a quite loose notion of error: a process is ill-typed if and only if it itself produces a message that it will always be later unable to treat. If a process does not work in a "bad", unyielding environment, the process is still well-typed. Other works by Colaço, Dagnat, Pantel and Sallé use abstract interpretation methods based on static set analysis [CPS97, DPCS00, CPDS99]. They apply it to an actor language and are able to detect "safety orphans", *i.e.* messages that can be detected as undeliverable in a finite time. Finally, Najm and Nimour [NNS99a, NNS99b, NN97] propose an actor calculus with private and public types with one-way (*i.e.* client/server) types featuring regular or infinite-state behaviors. Their type system detects all "message-not-understood" errors. Infinite-state types are obtained by adding an integer counter to each type. They propose a notion of type equivalence and subtyping based on (higher-order) language equivalence, which is efficiently solved by building bisimulations. Our type system essentially stems from the results of Najm and Nimour, but it adds a notion of time and obligation, that were absent from those previous results.

Other type system put types on processes, and not only on ports. Among early works, Boudol [Bou97a] proposes an analysis method based on linear logic for the blue calculus [Bou97b]. Due to the vew low restrictions imposed on the language, his type system is able to detect a small subset of "message-not-understood" errors. Works by Puntigam [PP99, Pun99, Pun97] are based on types that exhibit sets of named tokens that can be taken or put by processes when they perform a sequence of interaction. If not enough tokens can be taken, the demanding process is blocked. It is statically decided that all possible sequences will succeed. Various versions of the type system have been proposed, one of them with non-regular types [Pun99]. In [PP99], a solution is proposed to ensure the success of request/response exchange sequences. Such sequences could not be expressed in Najm and Nimour's works, but they are expressible in ours. Kobayashi *et al* [Kob02, Kob00, KPT99] propose types that denote sequences of interactions where non only the absence of deadlock, but also the absence of livelock can be detected. This is obtained by annotating action prefixes that must eventually succeed. Then, according to a *strong fairness* assumption, it is checked that all processes that wait ad-infinitum for a resource will enter infinitely-many times into a race condition. The strong fairness then guarantees infinitely-many accesses for each racing competitor. This work is close enough to ours, but it subsumes a rather abstract notion of time which is not liable to describe timed computations. Finally, we also have to mention the works of Yoshida [Yos02, Yos96] on *process graphs* that aim at giving guarantees about the termination of processes when encoding typed λ-calculi computations.

Other related and recent works adopting the behavioral types on processes often aim at encompassing previously-defined methods or to check more general properties. among them we can cite Kobayashi and Igarashi [IK01, Kön00], and Rehof and Rajamani [RR01, CRR02]. Some results with applications to behavioral type systems have been devised in other contexts, mainly in verification: Henzinger and Alfaro with different version of regular interface languages [dAH01b, dAH01a, dAHS02]. Applications by authors related to the previous ones can also be found in embedded

software [LX01, LNW02].

5.7.2 Assume/Guarantee Reasoning

Assume/guarantee reasoning was first used by Chandy and Misra [MC81]. It have been since then adapted in many different settings, the main distinction being between message-based communication and communication through shared variables. On the shared variable side, one may cite Jones [Jon83], Stark [Sta85], Stirling [Sti88], Stølen [Stø91], He and Xu [HX91]; a more recent survey can be found in [XdRH97]. On the communication-based side, besides the original proposal of Misra and Chandy, a formalization of the same proofs system can be found in [ZdRvEB85], while later assaults were given by Pandya and Joseph [PJ91] and Zwiers [Zwi89].

There have also been attemps that are independent of the synchronization method, essentially based on linear-time logics. The firsts were Barriger, Kuiper and Pnueli [BKP84], as well as Pnueli on its own [Pnu84]. Later came Abadi and Lamport with TLA [AL95, AL94, AL93], Jonsson and Tsay [JT95], and McMillan [McM99]. All those works, except the last one, allow only safety properties as assumptions. The more recognized theorem from Abadi and Lamport allows the introduction of liveness properties in the guarantee part, at the condition that the specified modules are receptive. This is related to other works for hierachichal developments of receptive processes, such as [LT87, GSSAL94, AH99]. The works mentioned above in this paragraph use only linear-time temporal logic. Some recent works allow to reason also about branching-time properties, checking simulation relations [HQRT98, KKLS00].

Assume/guarantee reasoning has also been accommodated to real-time specifications, by Hooman and Widom [HW89], Hooman [Hoo98], Abadi and Lamport [AL94] and de Roever *et al.* [DTDV96].

5.7.3 Conclusion

We have provided a compositional way of reasoning about safety and liveness properties of our processes. Those properties of well-typed processes can be used in *bottom-up* development, to produce a complete system by assembling many separate parts. In this regard, our type system forms a *module system* for π-calculus processes. After interface types have been written, each component can be produced separately. Type compatibility provides a powerful, decidable way of composing processes afterwards. Using enumerative methods, the check for type compatibility can furthermore be automated, since our types are finite-state. Type compatibility also offers a way of recovering the realizability property for processes composed from non-realizable part. It is indeed sufficient to enclose an environment-constraining configuration into a non-environment-constraining set of processes, to obtain a realizable process.

The notion of subtyping that we defined also allows *top-down* development methods. One may indeed replace parts of a specification with more refined processes, as long as those processes are of a subtype of the type of the processes originally in place. We however do not have experience on how practical our subtyping relation is. It seems *a prirori* very restrictive, and quite untractable in general, though.

We can also compare our behavioral type system to the previously mentioned works. First of all, the introduction of a behavioral type system for time-bound computations is unseen, to our knowledge. Only [Kob00] provides a very loose notion of discrete time for interleaving computations, and an accurate such notion in a truly-concurrent reduction system, where a clock tick is given at each parallel reduction step. The notion of offers and requirements expressed in processes and types also gives us a more flexible way of treating actions. In the type systems that we know, obligation is reserved to output actions. This is an implicit reason for which behavioral type systems have been applied to actor-like languages, since in asynchronous message-passing the emitter of the message has indeed the control of those actions, the implicit meaning of asynchronous emission being to force the environment to be ready to synchronize eventually.

Chapter 6

Applications

6.1 Introduction

A clear motivation for our work since the beginning was to provide a way of reasoning about distributed real-time computations in a formal and compositional manner. We first review the issues surrounding the notion of *contract* in computation, and we situate our proposal in the so-provided context. By the way we examine some of the applications that contracts have found in software engineering. We afterwards present an asynchronous variant of π^δ based on a more programming-oriented actor-like language. We show how this language provides a formal basis for the expression of model objects in real-time execution platforms (such as virtual machines or operating systems). Employing model objects is a part of the recent trends in middleware platform engineering. We also propose to use the proposed actor-like language as formal support in UML-RT specifications to write timed components.

6.2 Contracts in Logic and Computation

6.2.1 Contracts in Deontic Logic

The definition of *contract* can be better understood in *deontic logic*, the logic of law and normative systems [von51, MW93]. The salient feature of deontic logic is the ability to distinguish the *factual* world from the *ideal* world. In an ideal world, there are rules that *ought-to-be* respected; these rules can be tautologic, satisfiable, or antilogic. In the factual world, the rules of the ideal world can be contradictory to facts: everybody should respect the law, but there are people who do not. Deontic logic hence distinguishes not two but three levels of truth for a given proposition p:

- p *should* happen in the ideal world, noted $\mathbf{O}(p)$,
- p *may* happen in the factual world, noted simply p,
- p *should not* happen in the ideal world, noted $\mathbf{O}(\neg p)$.

Deontic logic is therefore the logic of *obligation, permission*, and *prohibition*. Both the ideal and factual world can be observed, and the occurrence of facts may induce the change of the rules in the ideal world as well as the occurrence of other facts.

Deontic logic has been used by Lee [Lee88] to automate the treatment of electronic contracting. A contract is a written document binding several participants that have *obligations* regarding one another. If one participant fails to respect its obligations then it can be deemed *guilty* of the fall of the contract. Other participants bound by the fallen contract are then freed from their obligations. Modalities available in contracts are therefore:

- *oblige*, making some non-obligated action obligated;
- *waive*, making some obligated action non-obligated;
- *permit*, making some forbidden action permitted;
- *forbid*, making some permitted action forbidden.

An essential aspect is then to be able to *compose* contracts. That is, it should be checkable if there exist a factual execution that satisfies all the (mutual) obligations present in the composed contracts.

We shall use deontic logic as a unifying formal basis to relate our approach to other ones in behavioral typing, assume/guarantee reasoning, and contract specification for distributed objects. We present all these methods as ways of introducing more or less powerful notions of contracts in computation.

6.2.2 Contracts in Behavioral Types

Behavioral types (also called behavioral interfaces) usually take the form of an automata or terms built using a pre-defined process algebra [RRV99, dAH01a]. The notions of obligation, permission and prohibition therefore remain implicit in behavioral types, since they may specify only factual properties. Types can however be considered as fully-fledged contracts by interpreting them in an adapted manner. In all the behavioral type systems that we know of (except ours), obligations are yielded by message sending: the environment of a process should never be able to block the delivery of a message for an infinite time. The notion of obligation is therefore tied to the generally accepted fact that processes should keep control over their output actions. This is an explanation for the many applications of behavioral typing to actor languages, where asynchronous interactions indeed ensure any process to send messages whenever it decides to, independently from its environment.

Behavioral types, as the processes they type, are neither input-enabled nor receptive [Dil89b]. When composing types, the consistency of the corresponding contracts (what we called *type compatibility* in the previous chapter) is therefore not true for all type combinations, and it must be checked as a pre-requisite to (process) composition. It consists in proving that no "bad state" is reached where the environment and the process do *interfere* [OG76] in an incompatible fashion; what is guaranteed is usually the "no-missed-synchronization" property. The compatibility test is based on a game between the types that are candidate to composition. The test yields a positive result if none of the two types is forced to breach the contract by disrespecting an obligation imposed by the other type. Types therefore have incompatible behaviors if the game ends into a deadlock. There are several theories of

transition systems that allow to distinguish "levels" of deadlocks, according to their cause [vG01]. The adopted semantics for types have therefore ranged over those theories, from reachability sets in trace semantics [Pun97], to variations on failure sets (defined in [Hoa85], used in *e.g.* [Nie95]), to testing-based semantics (defined in [NH83, BN95], used in *e.g.* [Kob02]), and to higher up in the branching-time spectrum.

The composition of given behavioral types (and of processes respecting those types) is then authorized only if the types are compatible. As type compatibility is a precondition to type composition, this approach is name *optimistic* in [dAH01a]. The pessimistic approach would require that types do not lead to an error when composed with *any* other type, as it can be done when processes are receptive [LT87]. The existence of a factual execution that satisfies all the type obligations can be seen as the proof of existence of some *friendly* environment that will prevent the compatible types from producing an error [RV00, Pun97].

6.2.3 Contracts in Hoare Logic and Its Extensions

Another approach to proving properties of program consists in explicitly formulating the obligations of each participant in a computation. This has led to many compositional and non-compositional proof methods dealing with interference in concurrent processes. Stemming from the works of Floyd [Flo67] and Hoare [Hoa69] on axiomatic systems introducing pre/post conditions and invariants for (sequential) control-flow languages, Owicki and Gries provided a non-compositional proofs system for invariant properties of interfering concurrent programs with shared variables [OG76]. A post-condition is an obligation (of total or partial correctness) that must be ensured by a process when it is executed from a global state that satisfy its pre-condition. If the pre-condition is not true when the process is called, the process is waived from terminating and ensuring the truth of its post-condition.

Pre and post conditions hence form a contract between a process and its environment: the clause in deontic logic describing the above situation resembles $Pre \Rightarrow \mathbf{O}(Termination \wedge Post)$. However those contracts are very *weak*, in the sense that they do not specify enough about the guarantees and assumptions provided by both the process and its environment. This weakness comes from the fact that pre/post conditions and invariants use predicate logic only, which does allow neither full temporal reasoning over executions nor direct expression of obligations. The verification method hence relies on checking that $\mathbf{O}(Pre)$ is true at any step for any pre-condition and any process in the specification. The full specification of processes must then be parsed to check this obligation, yielding non-compositional proofs. A formulation of Owicki and Gries' method for message-passing processes can be found in [AFD80], while a bridge between formalisms using shared-variable-based interaction and formalisms using message-based interaction has been established in [LS84].

Assume/guarantee specifications [AL93, AL95] are enhancement of Hoare triples dealing with temporal reasoning. They can be effectively formulated as Hoare triples [Hoo94, XCC94]. They allow for circular, compositional reasoning by providing contracts of the form: $Pre \Rightarrow \mathbf{X}\,\mathbf{O}(Post)$, where $\mathbf{X}$ is the usual *next* operator from

temporal logic [MP91]. This means that, if the pre-condition is respected in the current step, then a process has to make the post-condition true in the next step to be a model of the specification. The used language is therefore much more powerful than Hoare logics and behavioral types, since it can describe accurately the interferences that may occur between a given process and all the possible environments that can be composed with it.

Here again, the composition of two processes is conditioned by their compliance with one another's assumptions and guarantees. As in the case of behavioral types, assumptions and guarantees are separated by placing each action under the control of exactly one process [BKP84, AL95, XCC94]. All the actions not under the control of the process are under the control of its environment. Checking the compatibility of assume/guarantee specifications can be done by building a graph relating pre-conditions at step k and obligations of post-condition satisfaction at step $k+1$. If all pre-conditions at step $k+1$ are implied by the post-conditions at the same step, then performing mutual induction on models of those specifications is sound [MC81, AL95, McM99]. If oppositely there exists a cycle at level k or $k+1$ between two propositions, then mutual induction can not be performed safely.

To conclude, the distinction between assumptions and guarantees hence provides a fully expressive way of specifying contracts: assumptions are obligations on the environment, while guarantees are obligations on the considered process. this has been, at least to our knowledge, first noticed by Shankar and Lam in [LS91]. A later proposal aware of this fact can be found in [FNS97].

6.2.4 Contracts in π^δ

In all the technical presentation of the previous chapters, we have followed the usual way of presenting behavioral type systems. We have produced non-realizable processes, as well as non-realizable types that constitute finite-state abstractions of our processes. We have shown that the absence of deadlocks in configurations containing two types put in parallel is sufficient to prove the absence of missed synchronizations in parallel processes (in our case *Error* deadlocked processes).

Yet, we have given a deontic flavor to our specifications. By placing offer and requirement sets on time-passing transitions, we allow processes to observe and take into account not only the *factual* aspects of their behaviors, but also the *ideal* ones. Processes therefore have a way of exhibiting obligations to their environment and to know whether the expected behavior has been obtained or not. To maintain a conformance with other behavioral type systems and in regards to proposals we shall make in the next section, we let processes adopt a *fail-stop* behavior in the case where an obligation has not been respected. We have made this choice mainly because of semantical issues, and also because it suits well to a language intended to represent abstract computations in a quality-of-service-enforcing distributed platform. We have found yet a suitable solution to do otherwise. In such a case, a continuation would always be well-typed in our system whenever an urgency condition would fail to be satisfied; this corresponds to waiving a component from respecting any obligation after the behavioral contract on the incriminated port has fallen.

As stated when presenting the language, our processes have a greater expressive

power when it comes to specify the control over actions. We can now see that this is due to the possibility of placing obligation modalities on any action (silent, input or output) using urgency conditions. We finally note that we have been able to produce assume/guarantee reasoning when type-checking parallel processes. This is because urgency conditions have strict upper-bounds, and that letting time pass to the next region defines a clear cut between two steps, allowing to break circularity in the reasoning whenever possible.

6.2.5 Contracts Elsewhere in Computing

There have been many study of contracts in the recent years, both in informal and formal settings. As stated above, all behavioral type systems and assume/guarantee specifications can be seen as employing contractual forms. We shall merely give some references in the domain of software engineering and theoretical computer science.

First, the advent of object-oriented and component-oriented computing as well as the outcome of multimedia systems has led to a search for more informative notions of interfaces. A well-known pioneer in the domain is Bertrand Meyer, who proposed a *contract-oriented* design method [Mey89, Mey92, Mey91]. Each method is then adjoined a Hoare-triple-like invariant, pre and post-conditions, that are checked for validity at run-time. A formal study by Liskov and Wing [LW94] allows for subtyping in object-oriented specifications with similar pre/post conditions.

A behavioral notion of contract has been given in [HHG90], where components bear *type constraints* and *ordering constraints* on events, taking the form of an `instantiation` (for the initial conditions) and an `invariant` directives. *Coordination languages* [GC92] and *architectural description languages* [AG94] have also been a wide-open field of investigation for the notion of contract and interface. Among those works, we should cite [AF99], that proposes a notion of contract based on superimposition, as are architectural connectors. Several other informal proposals have been made for component-based software development, among which we may cite [Hol92, BJPW99, Gie00]. In the next section, we will add to this picture other (more specialized) contract-oriented languages, that are used for quality-of-service (QoS) specification.

A formal study of contracts has been provided by Back and Von Wright in the *refinement calculus* [BvW98, BvW00]. Back and Von Wright split the variables of a system under process variables and environment variables, while they introduce *released* and *never-satisfied* processes in their calculus, that respectively represent the breaching of a contract by the environment of a process (the process is then freed from respecting any of its engagements) and the breaching of a contract by the process itself. Some algebraic laws are given, as well as a rule for sub-contracting.

The component-based interface theory of Alfaro *et al.* [dAH01b] allows the designer or programmer to tag each method with a set of prohibited methods. The transitive closure of the set of method calls is also built, and both sets are compared to determine the consistency of the so-defined contracts.

Finally, the translation of input/output automata in an algebraic setting [Vaa91, Seg92, Seg97] have yielded different notions of contract violation. Indeed, as processes may not be input-enabled although the underlying I/O automata are, the

unpredicted incoming messages must be treated in some way. Vaandrager [Vaa91] chose the *angelic* approach, while Segala [Seg92] chose the *demonic* approach; in the angelic approach the process does not change, while in the demonic case, the process may adopt any possible behavior, becoming the divergent process Ω. The first way can be seen as forbidding a process to escape from its obligations whatever the behavior of its environment is, while the other way allows a process to be freed from its obligations whenever its environment behaves badly.

6.3 Provable QoS Support for Middleware Platforms

Preserving properties of a timed specification when stepping to implementation is currently an open issue. It has recently been the goal of two separate efforts, respectively in formal methods and operating systems/middleware communities.

In the formal methods community, the concern was to provide *implementable* languages; languages for which there may be a compilator and, if not an efficient, at least an acceptable run-time execution support [SHG89, FHW99]. We do not know whether π^δ programs could retain such an efficient transformation towards an actual binary format. Due to the simplicity but also of the expressiveness of the language, we conjecture that the answer would be negative. An adaptation of the original model would be necessary, but the presence of time-related actions would prevent us from providing a simple solution as it has been done in Pict [PT00]. We think that, oppositely, our language may actually be of noticeable help to rule and coordinate computations performed under the control of some middleware platform or virtual machine.

Several separate efforts have indeed been led regarding platform support engineering for the execution of both hard and soft real-time constrained tasks: Nemesis [LMB+96, MLM94], Polka [DHS98], DJINN [MNCK99], Tao [Sch00], dimma [DFH+98] and Jonathan [DHDTS98], to name a few. They all require the user to specify what can be interpreted as *QoS contracts* [CBS+95] between entities in the system. Those contracts are used at run-time to produce executions that tentatively share available resources in an "optimal" fashion, and respect the terms of the contracts given by all the currently executing tasks. Each implementation has however its own definition for optimality: this yields policies that range from strict admission control to fair degradation between tasks. The cited platforms achieve reasonable results for regular load conditions, the average level of those results depending on the crucial parameter in real-time systems that is *predictability*. As an example, in Tao [Sch00], scheduling must in general be feasible *off-line*, before the execution of the program start. This is a very strong constraint, indeed. Authorizing more freedom and potentially evolving conditions such as in Polka [DHS98] implies much lower predictability, that may lead to the non-respect of timeliness conditions such as deadlines.

Our goal is not to present policies or new methods to perform resource allocation in distributed real-time platforms. We merely suggest that, in very open and changing environments, using a formal language such as π^δ for coordination and anticipation in resource allocation appears to be a reasonable solution.

6.3.1 QoS Specification

Many language have been proposed for quality-of-service representation. The goal in most cases is either to introduce QoS annotations during software development or to negotiate QoS parameters at run-time. Some QoS-languages therefore retain a formal-enough semantics to undergo static analysis for consistency or correction.

In the software engineering world, the QML proposal [FK98] aims at completing UML class diagrams with QoS annotations so that these constraints can be taken into account as early as possible in the development process. QML specifications introduce contracts and contract types as a way of placing those annotations. Contracts include values that range over ordered domains. Contracts also specify constraints on those values, and it is checked that a contract does not contain any inconsistencies. A similar proposal has been made in the *intelligent network* community [AS01], as (meta)-class stereotypes for a UML service creation profile. Also related is the QuO framework for quality-of-service in objects [LBS+98, LSZB98] that defined a *contract description language* (CDL). They illustrate the use of contracts over objects at run-time to regulate the execution and enforce timeliness. Finally, several proposals by Eliassen *et al.* introduce a language that allows to write interface types in terms of high-level QoS parameters [EM98, RE00]. Consistency checking can be done on those interfaces through an ad-hoc type-checking procedure.

All the works above rely on a *static* notion of QoS specification: a QoS environment is represented by a vector of variables that each retain some value. We now present contributions adopting the other approach, where the desired or enforced QoS is given by the *behavior* of some computational entities. We start by Frølund and Agha [FA95], who proposed to coordinate actors through external *synchronizers*. A synchronizer is a special actor that "completes" static actor interface by observing interactions occurring at those interfaces, and by forcing those interactions to respect a certain order. A later timed extension allows to specify real-time constraints [RA95, NA99]. The advantage is that synchronizers can be assembled easily. The mentioned contributions however do not provide any way of statically detecting inconsistencies among synchronizers. A related approach is the one adopted by Février and Najm [FNS97, FNLL96], where *observers* are used to specify contracts put onto *binding objects* [ODP95]: an observer may not constrain the behavior of objects in the system, but it may signal that an obligation has not been respected at a given interface. The repercussions of such an error signal depend on the behavior of the system itself, that is if there exists in the system a controller object that is able to treats error messages in an adequate fashion.

In general, any real-time behavioral formalism can be used to represent QoS constraints; this includes timed process algebras (as in the two previous examples), and timed logic. Among process-algebraic languages, RT-Lotos [CdO95, SSdSC98, SC00] has been particularly devised and used to model *hypermedia* documents. Those are documents composed of several parts where text, sound, and video are organized so that they are displayed or reproduced to the spectator in a seamless fashion. The tool RTL[1] allows to simulate and formally verify RT-Lotos specifications through model-checking. On the logic side, we may cite the use of first-order logic with quan-

[1] available at `http://www.laas.fr/RT-LOTOS/index.html`

tification over time, as used in the QL [BS97] logic. Similar uses of first-order logic have been provided by Hooman in his proof systems [Hoo94]. Timed temporal logics in general are able to express hard real-time QoS constraints, and we therefore can not give an exhaustive list of such uses. An example of use of temporal logic with the precise idea of representing QoS constraints in mind has been given by Stefan Leue [Leu95].

6.3.2 Model Components and Type Checking

We follow recent trends in platform engineering that advocate the use of *many levels* of objects into distributed computations. In particular, Mitchell *et al.* [MNCK99] separate objects between *active objects*, that carry over the computation performing the real job, and *model objects* that only coordinate and control the computation. Only model objects contain information about the wanted or enforced QoS constraints. It is their duty to ensure the maintain of timeliness in a time-constrained environment. This architecture is used to provide an adaptive middleware that allows seamless adaptation in resources allocation when execution conditions change.

The goal in the above work is to reach separation of concern and efficiency. As model components assemble all the time-related and reconfiguration-related information, active objects are freed from this intricate task, and they shall behave in the same way whether there are actual timing constraints or not. In [MNCK99], model components are written in Java. We also think that model object should have behaviors, because using contract languages such as CDL [LSZB98] or the language proposed in [EM98] does not indicate exactly which actions are performed neither the time when they are performed. We propose to use a more formal language such as π^δ; the RT-Synchronizers of Nielsen, Ren and Agha [NA99, RA95] have also been devised to fulfill such ancillary tasks. Being more application oriented, RT-synchronizers have however not been equipped with static analysis methods that would allow behavior predictability in an accurate fashion.

A related approach concerns *Model Carrying Code* [SRRS01]. There, the goal is more to prevent illegitimate resource access by specifying those accesses formally, and by controlling that the effective accesses are a subset of the authorized accesses. The model components are interpreted as a token of the existence of a contract between programs and the underlying execution platform. If a program does not comply with its model, it breaches the contract, and the platform, although it has accepted to execute the incriminated task an to provide it a certain QoS, may take all necessary reprimanding actions.

If any program (that is, *code*) that demands execution to a middleware platform comes with an π^δ model of its behavior, the execution platform is by the way provided with an additional mean of observation and control over the executed programs. This is a way of avoiding *Denial of Service* (DoS) attacks, that occur when some process floods another process with requests so that the latter process may not properly function. By doing type-checking over the given π^δ specification, the platform could assure itself that all the launched processes exhibit only compatible interactions, and that, according to a chosen access control policy, there are sufficient resources on the computer in terms of CPU time, memory, network bandwidth, etc. The

type-checking phase corresponds to a replay of a proof of *non-blocking* behavior; it may be completed by checking that all types are *strongly non-zeno*, so that a process may not require the access to an unlimited number of resources in a finite time. This could be done also by directly adjoining the proofs of well-typing and non-zenoness to the specification. This approach has been called *proof-carrying code* [NL96]. After all necessary checks have been done, the platform may then freely discard a process that does not fulfill its obligations, or that overpasses its rights (for example in the goal of performing a DoS attack). A similar proposal has been done recently by Teller *et al*, using the *ambient* calculus [TZH02].

6.4 Integration with UML-RT

We now provide a way of integrating our work with the UML-RT method [RS, Sol97]. Advances in the use of formal methods are usually obtained by either providing formal semantics to informal design languages such as Flowgraphs [GBSS98], or by integrating formal aspects in the software development process using both formal and informal languages. Therefore, proposals pushing formal methods into UML are many, and some of them have been cited here before [AF99, FK98, AdSSL+01]. Our contribution in the remaining of this chapter is very informal, and it has not been pushed very far. We only present the basic ideas that motivate and permit a natural integration of our method in a development process using UML-RT as a notation.

In this section we employ an asynchronous version of our calculus, that we think is best suited in general for programming or specification. That is, the primitives of the language have the same expressiveness, but their formal writing is more simple and more agreeable to the eyes. The reason for this choice is that actor-like languages seem best suited for programming than the rather elementary and rapidly inadequate notations induced by π-calculus-like languages. We hence now present this asynchronous, simple actor-like version of π^δ, that we shall name here $\mathcal{A}$rtOC.

6.4.1 $\mathcal{A}$rtOC: An actor-like, asynchronous π^δ

We shall expose the semantics of the calculus only informally. The results that we obtained on the synchronous calculus carry over to the asynchronous calculus rather straightforwardly. As in the synchronous case, our actors are augmented with the capacity to test and reset clocks that we call here *timers*. Specifying timing constraints is done by placing guards on the values of timers. As mentioned in the section on behavioral typing in this chapter, the asynchrony hypothesis ensures the control of all input actions to the processes that perform them. Urgency conditions become much less useful in that context; we therefore relinquish them for the sake of simplicity.

Hence, all contractual obligations are borne by message sendings. The other main difference is in the type language. Due to the time taken by messages to go through the network, we have to limit the way in which the sense of circulation on a given port may vary in time. For this reason, we introduce two-part types, where

each type has one control and one stream part. The semantics and the type theory behind the language are exposed in [BNSL00].

Actors [AMST97] are self-evolving objects that communicate by sending signals (usually called *messages*) through asynchronous channels. An actor may *send* a message (following in a *non-blocking* policy), wait for a message, adopt the behavior of (*ie. become*) another actor, or *spawn* another actor. A timer allows to specify timeouts corresponding to infinite or finite, deterministic or non-deterministic, delays. This yields the power to express also *non-blocking* and *finitely-blocking* message receptions. Our actor communicate through ports, that are names which can be passed around among actors exactly as in the π-calculus. The communication topology may therefore be modified during the execution.

C	::=	$B \mid C \mid C$	G	::=	$\delta \leq x \leq \nu$
Dec	::=	$A(\tilde{p}) = B$	B	::=	∞
	$\mid$	$Dec\ Dec$		$\mid$	**new** $a : T$ **in** B
p	::=	$u : \rho\, T$		$\mid$	**become** $A(\tilde{a})$
ρ	::=	$! \mid ? \mid !! \mid ??$		$\mid$	**spawn** $A(\tilde{a}) > B$
ν	::=	$\infty \mid \delta$		$\mid$	$!v.s(\tilde{a}) > B$
δ	::=	$Integer \mid \delta + \delta \mid \delta - \delta$		$\mid$	**timer** $x = \delta$ **in** B
R	::=	$?a\ [s_1(\tilde{p}_1) > B_1 \ldots$		$\mid$	**if** G **then** B_t **else** B_f
		$\ldots s_n(\tilde{p}_n) > B_n]$		$\mid$	**timeout**$(G) \sum_{i=1}^{n} R_i > B$
				$\mid$	**timeout**$(G) > B$

Figure 6.1: The Syntax of $\mathcal{A}rtOC$

Table 6.1 details the $\mathcal{A}$rtOC syntax. In that figure, C stands for a configuration, Dec for a set of declarations, B for a behavior, A for a named actor, p for a formal parameter, ρ for a role, x for a timer, G for a guard on one timer x, a for a port name, T for a type name, and R for the reception on a given port. In that context, $\tilde{u}$ represents the list of names $u_1 \ldots u_k$. Finally, we will also use $\mu ::= ! \mid ?$ to name the action prefix of sending or receiving a signal.

A set of declarations Dec forms a name context for any configuration C. C is a set of behaviors put in parallel. An actor declaration is written $A(\tilde{p}) = B$, which associates name A to behavior B with formal parameters $\tilde{p}$. The syntax $u : \rho\, T$ associates role ρ and type T to port name u. Each port is said to play a particular role regarding a given service. Roles can be *client* (noted !!) or *server* (noted ??), *source* (noted !) or *target* (noted ?). The source and target roles are *private* endpoints bound by a QoS contracts. Their features are similar to our behavioral types for synchronous interactions: they describe when messages can be sent, and when a process owning such an interface should be listening on the port. As in the synchronous case, source and target roles exhibit non-uniform services. We therefore maintain the rule that a source role can only be paired with one target role, and conversely. Each source-target couple forms an exclusive binding interaction group; both bound processes can emit and receive signals over *reliable FIFO channels with fixed minimum and maximum transmission delays.* Ports with

client or server roles correspond to *uniform*, *untimed* services. They are "traditional" IDL-like interfaces, defined as a set of signals with arity and type for each of their formal parameters. A server can then concurrently receive messages from *many* clients. In the remaining developments we will often identify roles and ports, writing "role ρ" instead of "port of role ρ". The distinction should usually be clear from context. In any case, the type of a port will be described as a timed automaton.

We now present one by one the possible actions of an actor. **new** $u : T$ **in** B creates two new ports named u of type T and then behaves like B. According to the specification associated to type name T, either a client and server roles or a source and a target roles are created. An actor can behave as another actor: **become** $A(\tilde{u})$ behaves like A with ports $\tilde{u}$. **spawn** $A(\tilde{u}) > B$ creates a new actor that behaves as A, passing the effective parameters $\tilde{u}$ as arguments; A is then executed in parallel with B. $!v.s(\tilde{u}) > B$ emits a signal s through port v with arguments $\tilde{u}$ and then behaves like B. A timer is simply introduced with the instruction **timer** $x = \delta$ **in** B. **if** G **then** P_t **else** P_f assesses guard G, then executes P_t if it is true, or P_f if it is false. A guard simply checks if the value of a timer is included in some interval.

A reception may feature a choice between many ports: **timeout**$(G) \sum_{i=1}^{n} R_i > B$ waits until G becomes true before executing B, but it can be interrupted before that whenever a signal delivery in one R_i occurs. Each R_i indeed comprises the name of a port u to listen to, and the set of expected signals along with triggered behavior when the reception occurs. Finally, **timeout**$(G) > B$ waits until G becomes true, and then behaves like B.

The binding rules describe how references are forged and how they can be sent away to set a service up. Roughly, the rules follow the ones given in the synchronous case. For private ports, the binding is said *explicit*: both roles are created by the same actor, and this actor establishes the binding by sending the target role. Once it has done so, neither role can be given to another actor. This scenario is usually referred to as *first-party* binding. The binding rules for uniform services are slightly different: both client and server roles are created by the same actor, but it is then free to send both roles away. That is, an actor does not lose the capacity to use a client role when this latter is sent away; client roles are always duplicated, and any number of them can concurrently send signals to one server role. However, only one server role may exist for a uniform service, and when an actor sends away a server role, it loses the ability to receive signals through it.

$\mathcal{A}$rtOC configurations can be analyzed much in the same way that π^δ configurations did. A crucial point here is the non-zenoness of processes. Indeed, the type compatibility is analyzed by performing reachability analysis on a system comprising the two types and a sufficiently large, finite channel. If one of the processes (or types) is zeno, then we can not perform reachability analysis, since the content of the channels can grow unboundedly. A system with at least one unbounded channel is Turing-expressive [BZ83]. The main ideas for type compatibility verification and type checking are given in [BNSL00].

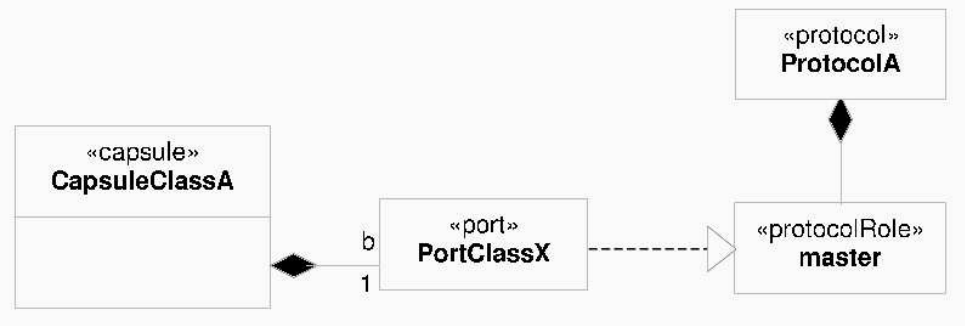

Figure 6.2: The Relationships Among UML-RT Entities

6.4.2 A Meta-Architecture For UML-RT

We first introduce the UML-RT notation [RS], an offspring specification language stemming from the ROOM method [Sel96]. The UML-RT notation fosters the clean separation of computing entities into components linked to one another by typed channels. After this short introduction, we provide some background information on UML models and meta-models. Then we show that there is a direct correspondence from the UML-RT modeling concepts to the elements in our theory of typed asynchronous timed processes. We close those developments by a short, illustrative example.

6.4.2.1 Unified Modeling Language for Real-Time (UML-RT)

There are mainly three elements in UML-RT specifications:

- capsules,
- ports and protocols,
- connectors.

The relations between those notions are described by the UML class diagram presented in Figure 6.2, taken from [GBSS98]. A protocol is a description containing all the legal uses that can be performed at some interaction point. Each use is called a *role*, and is itself an ensemble of legal behaviors allowed by the role. On figure Figure 6.2, one may say that protocol `ProtocolA` *owns* or *aggregates* the role `master`. A port class is a pattern over which ports may be instantiated. A port class *realizes* or *implements* a protocol role. A capsule contains computational entities that communicate through ports and must respect their protocols. A capsule class, over which capsules can be instantiated, may own several instances of one or many port classes.

In Figure 6.3, an example UML-RT collaboration diagram is given that use the entities we just defined. It also introduces *connectors*, that are the solid lines traces between ports. ports are the squares filled in black or white that are situated at

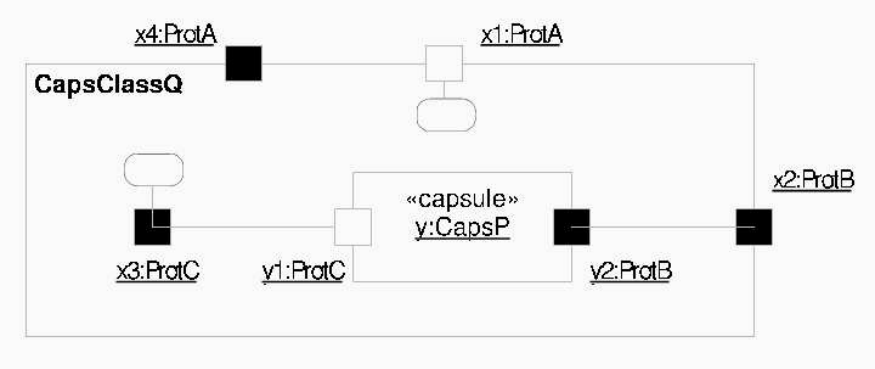

Figure 6.3: An UML-RT Collaboration Diagram

the frontier of capsules. In a collaboration diagram, only instances of classes are described. Therefore, ports are given names, and the type they are tagged with is the name of their protocol class. As one may imagine from the figure, connectors are purely *passive* links between ports. However, they can receive a behavior through association classes, upon which they can be seen as *interceptors*.

We now give a quotation from [GBSS98], that situates the contribution we may have regarding UML-RT specifications:

> [...] connectors [...] interconnect ports that play complementary roles in the protocol associated with the connector. [...] the protocol roles [...] have to be compatible with the protocol of the connector[1]. [...]
>
> [1] We will not discuss the rules of protocol compatibility here except that it is based on behavioral sufficiency.

As we have seen in the chapters 3 to 5, we have provided a way of expressing protocols and to check their compatibility. In that sense, our proposal exactly fulfills the needs expressed in the above sentence. We are now left to show how we may articulate the integration of our proposal to software engineering using UML-RT.

6.4.3 Notions of $\mathcal{A}$rtOC in UML

To make a pertinent contribution, we first have to decide the "right" level in UML-RT notation at which we should introduce the elements of our theory of timed actors. The "stack of models" is represented in Figure 6.4. The UML meta-achitecture (or *meta-model*) describes the relations existing between elements of UML notation in the UML notation itself. From the meta-model elements, *concrete* modeling concepts are produced by instantiation: at the application model level, manipulated entities such as classes, relations, *et cetera*, are *instances of* their description at meta-model level. At the application level, those notions are employed to produce actual applicative models. Modeling with UML-RT hence takes place at the same level,

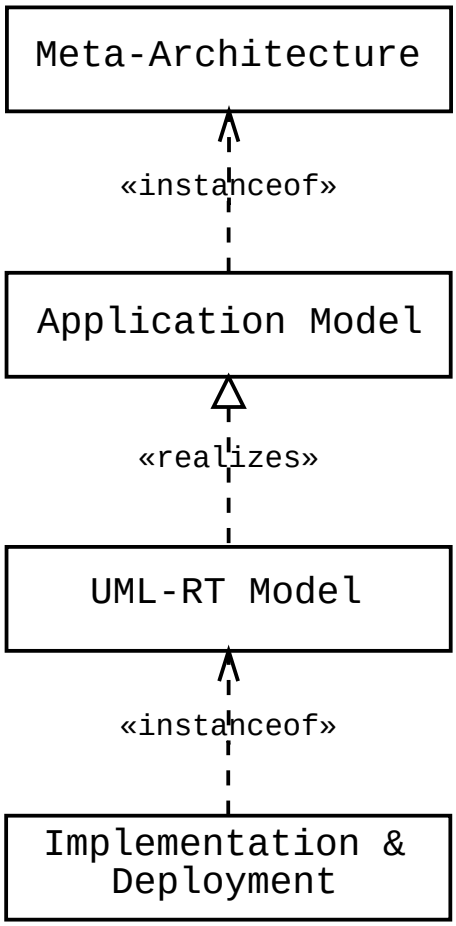

Figure 6.4: The "Stack of Models"

but the application level notation is enriched to make notions specific to UML-RT representable. This is done through an activity called *stereotyping*, which goal is to provide a mapping from specific elements in UML-RT specifications to corresponding elements at the meta-model level. If a proper mapping is provided, then an UML-RT model is said to *realize* a simple UML model that would not use any specific notation. The last instantiation relation intervenes between a model and its implementation.

Our contribution therefore consists in the introduction of $\mathcal{A}rtOC$ concepts at the meta-level, and their stereotyping that yield corresponding notions at application level.

The general links relating $\mathcal{A}$rtOC notions are described in Figure 6.5. We introduce the notion of *location*, though it was not present in the syntax of the language. Having locations is in our case just a syntactical convenience for grouping processes, that dispenses the programmer from providing channel characteristics (delay and jitter) for every possible pair of objects: the channel characteristics are specified between locations, and those characteristics apply to all objects of each location.

So, all the classes of the diagram in Figure 6.5 are tagged to be metaclasses. The metaclasses `ActorType`, `LocationType`, `ChannelType`, and `ContractType` can

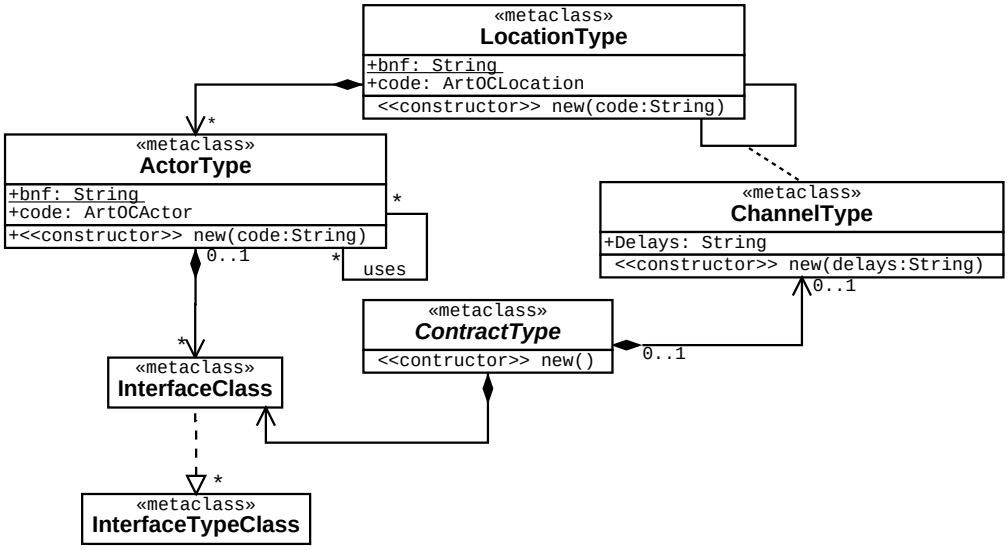

Figure 6.5: The Relationships among *ArtOC* Meta-Architectural Concepts

be seen purely as containers that enclose textual descriptions of the corresponding textual elements in $\mathcal{A}$rtOC programs. Those meta-classes will be used for stereotyping purposes, allowing one to give them precise graphical descriptions (using special symbols, graphs, boxes, etc.). Neither locations, actors, contracts, nor channels are first-class entities in $\mathcal{A}$rtOC: they do not exist at execution time, since they consist only in syntactical descriptions that disappear at compile-time. This explains the difference between the treatment applied to those entities (for which we shall only manipulate their types) and the one given to interfaces, which do exist at execution time.

The type of a location has two attributes: a description of the grammar of the $\mathcal{A}$rtOC language that tells how locations should be written, and a code for the location type, that is set by the constructor `new(code:String)` when a location type is created. Using the grammar, the constructor may check the syntactical correctness of the given code for a location type. The aggregation link between the `LocationType` and `ActorType` classes means that each location of a given type may instantiate actors of some given types. Each actor type in turn may execute the code of another actor type by executing a *become* instruction. This is shown by the looping *uses* relation from actor types to actor types. Let us now look on the other side to the attributed association from location types to location types. Each such association comes with a channel type, indicating the QoS characteristics (delay, jitter) of the channels linking objects in the two locations.

A contract type may reference zero or one channel type. This is necessary for private contracts, where the characteristics of the transmission must be known at compile-time in order to ensure type compatibility and perform type-cheking. Now, a contract type may enclose any number of interface classes (*i.e.* protocols) to build

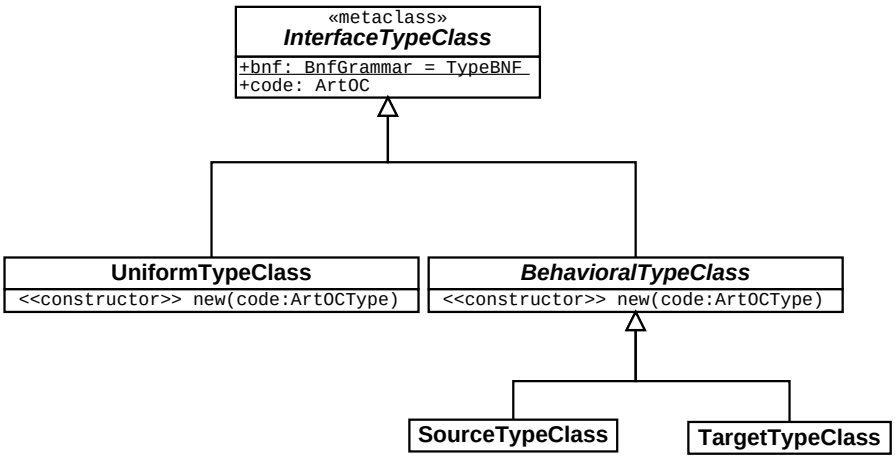

Figure 6.6: The Interface Type Hierarchy

its contract. In practice however, $\mathcal{A}$rtOC contracts are limited to two interface types maximum, since only bipartite contracts are allowed. An actor type may also reference any number of interface classes, and use them in the actor's $\mathcal{A}$rtOC code. Finally, the diagram states that an interface class is a realization, or an implementation of the more general class of interface types, that contains all the interface classes that can be written using $\mathcal{A}$rtOC.

We now review the interface type hierarchy, shown in Figure 6.6. Then again, instances of those classes contain the code of the protocol they should implement. The code can be syntax-checked during the instantiation process using the `BnfGrammar` attribute of the class `InterfaceTypeClass`. This class can be specialized into the classes of uniform types and behavioral types, respectively. The class of behavioral types can again be specialized into the class of source types and the class of target types.

Please notice that we did not represent the specialization of the uniform type class in client and server types, since this is of little interest here.

The contract type hierarchy is given in Figure 6.7. A contract type has no behavior on its own, it is used only as a way of grouping (hopefully) compatible interfaces. Contract types can be either implicit or explicit (see Section 2.1.2.3), according to the nature of the interfaces they bound. If the binding procedure is implicit, then the types of the bound interfaces must be uniform. There is an implicit contraint that can not be expressed directly on the diagram, but that can be written in the *Object Constraint Language* (OCL): in a contract rule under implicit binding, there can be only one server for as many clients as wanted. In the case where the binding is explicit, then there must be one and exactly one source interface and one target interface, respecting the exact $1-1$ cardinality.

To be easily usable in UML class diagrams, we must provide a way of graphically distinguish locations, actors, interfaces, etc. This is done by stereotyping the meta-classes. We apply the usual convention that, when someone writes a class diagram, he or she writes for example `Human` as name of the class, and not `humanClass`.

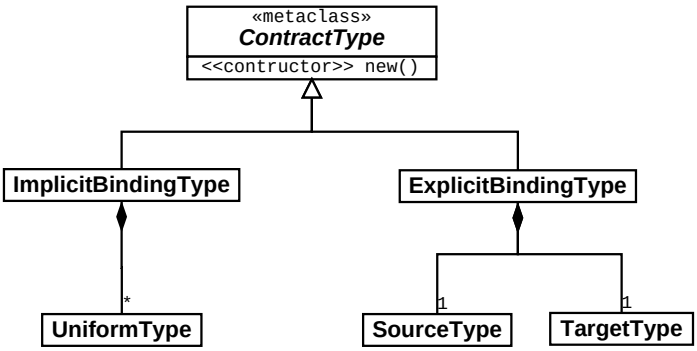

Figure 6.7: The Contract Type Hierarchy

Similarly, one shall write `Interface` at the application model level to write interface classes. Therefore we name `Interface` the stereotype of interface classes.

We do not give precise notations for our elements, which would be of little relevance here. Each element has however a set of tags, that are there to precise its properties. For example, an `actor` has a `code` tag, that precise the code for the actor. The other details regarding the stereotyping activity are quite straightforward; we shall now skip all remaining details and start to study the links between UML-RT and $\mathcal{A}$rtOC.

6.4.4 Relationships with UML-RT and Examples

We claim that there is an obvious correspondence between $\mathcal{A}$rtOC and UML-RT concepts. This claim is (we think) strongly supported by the diagram on Figure 6.9. This diagram shows on the right elements taken from the UML-RT specification language, and on the left elements that intervene in $\mathcal{A}$rtOC specifications. The arrows going from the right to the left are *realizes* arrows. It means that any element on the right can be implemented using an element on the left. The only difference is related to connectors, at the bottom of the diagram. The difference between connectors and channels is that our channels are purely passive entities, only containing delay and jitter information. Connectors may oppositely have behaviors when an association class is used. We have no possibility of straightforward correspondence in this case. Hence, the correspondence is not perfect, but it still seems convincing enough to us.

We are now going to treat as an example the specification of a video-on-demand transmission between a source and a target locations on a network. The diagram on Figure 6.10 shows that an emission center may have a relation with many clients through associated cable network channels. The paced emission of data is assured by a video server, that establishes explicit bindings with clients through interfaces with type AV_s (standing for Audio/Video source).

We shall not provide the formal semantics for types nor actors, but here we present the type of the receiving interface $\mathbf{AV_t}$. Its behavior can be described as:

- it receives audio and video at constant pace, one video frame each 40ms, and

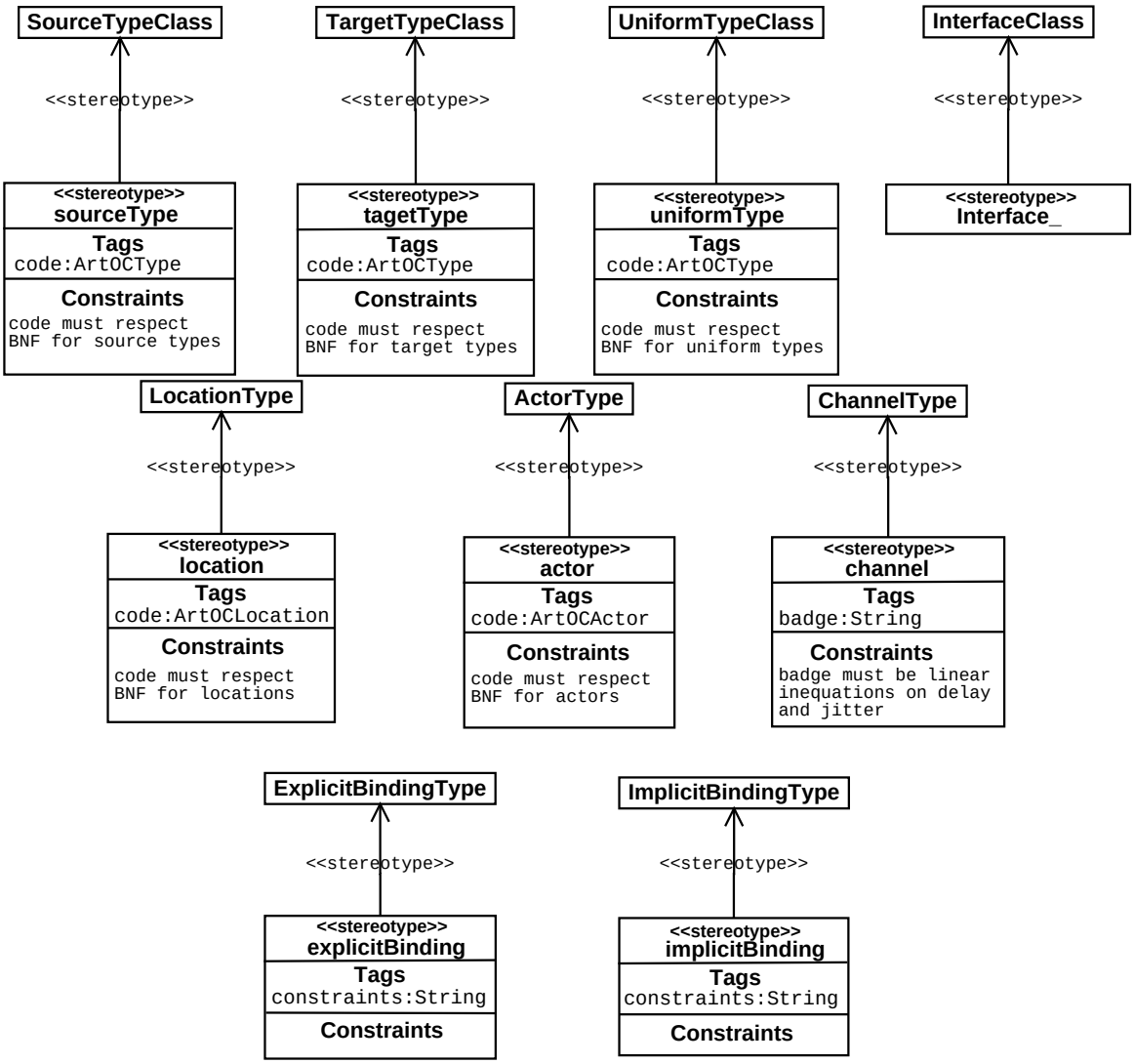

Figure 6.8: Stereotypes for $\mathcal{A}rtOC$ Specifications

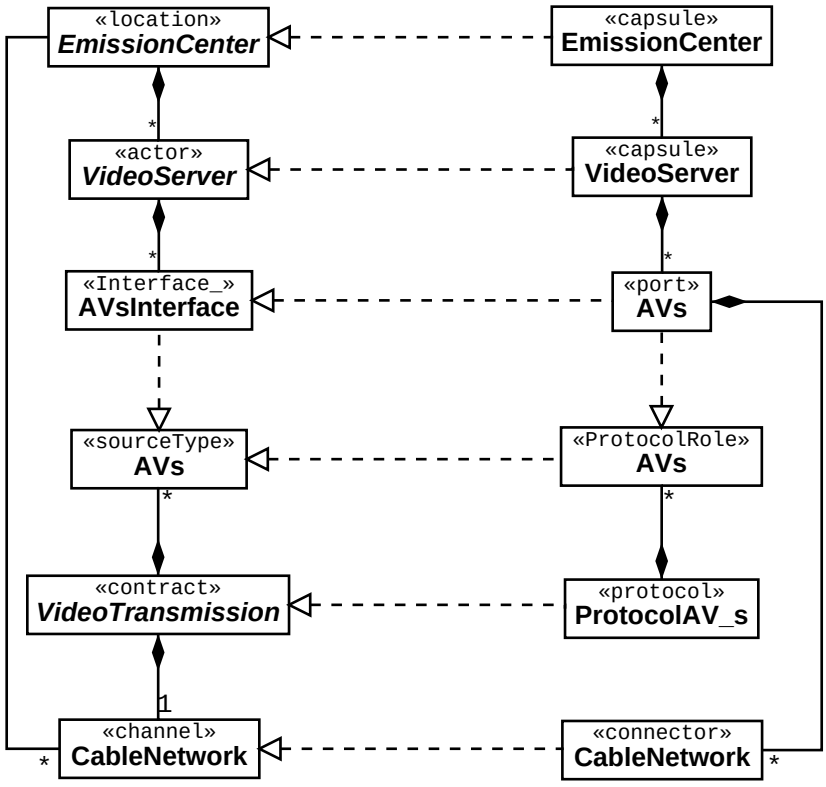

Figure 6.9: Realization of UML-RT specifications by *ArtOC* Programs

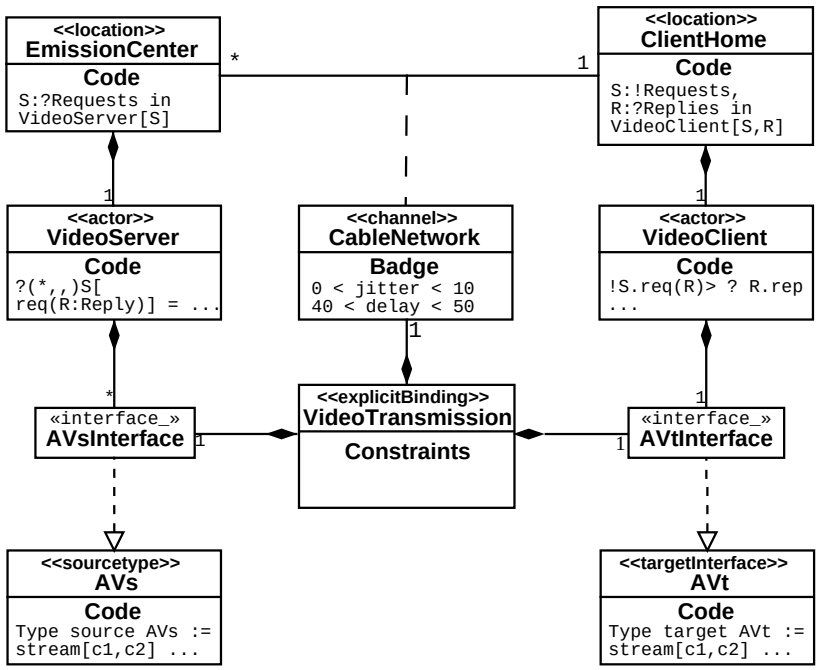

Figure 6.10: A Video Transmission Example in $\mathcal{A}rtOC$ UML notation

one audio frame each 100ms.

- The minimum delay for the transmission of a frame is named τ_0, and the maximum delay $\tau_0 + \tau$.
- Once the target interface has been received by the client, this one has 5 seconds to ask the start of the video, and then it must wait at least 5 seconds before it can ask to stop. The transmission effectively stops at most one second later.

$$
\begin{array}{lll}
\text{Type target } AV_t(\tau_0, \tau) := & & \\
\text{stream}[c_1, c_2] : & x_1 = \emptyset & \\
& x_2 = c_1 \text{ in } [40 - 2\tau, 40]\ \mathit{video}() \text{ until } 40\ ; \text{stream}.x_2(c_1 = 0) & \\
& \quad + c_2 \text{ in } [100 - 2\tau, 100]\ \mathit{audio}() \text{ until } 100\ ; \text{stream}.x_2(c_2 = 0) & \\
\text{control}[c_3] : & x_3 = c_3 \text{ in } [0, 5000]\ !\mathit{start}(); \text{control}.x_4(c_3 = 0) & \\
& x_4 = c_3 \text{ in } [0, 5000]\ ?\mathit{start\_ack}(); (\text{stream}.x_2, \text{control}.x_5)(c_3 = 0) & \\
& x_5 = c_3 \text{ in } [5000, \infty]\ !\mathit{stop}(); \text{control}.x_6(c_3 = 0) & \\
& x_6 = c_3 \text{ in } [0, 1000]\ ?\mathit{stop\_ack}(); (\text{stream}.x_1, \text{control}.x_7)(c_3 = 0) & \\
& x_7 = \emptyset &
\end{array}
$$

We also present a code for the video-on-demand location that contains the video server. There are three named behaviors in the location: Period_AV, Stop, and Video_Server. The server initially lauches the Video_Server behavior, passing it a public port S as argument. The server then waits indefinitely on that same port, waiting for new clients. In case such a client arrives, the server creates a new port

named Chan, and sends it to the client. Once this is done, the recursive behavior is started after some timing adjustments by calling behavior Period_AV.

```
Loc Video_On_Demand =
   S: Incoming_Req,
   Period_AV [ Chan: AV_t ] =
      ?(40,,) Chan [stop = Stop[Chan]] > !Chan.video() >
      ?(20,,) Chan [stop = Stop[Chan]] > !Chan.audio() >
      ?(20,,) Chan [stop = Stop[Chan]] > !Chan.video() >
      ?(40,,) Chan [stop = Stop[Chan]] > !Chan.video() >
      ?(40,,) Chan [stop = Stop[Chan]] > !Chan.audio() > !Chan.video() >
      ?(40,,) Chan [stop = Stop[Chan]] > !Chan.video() >
      Period_AV [Chan] ,
   Stop [ Chan: AV_t ] =
      !Chan.stop_ack() > 0 ,
   Video_Server [ S: Incoming_Req ] =
      ?(∞,,) S [ req (R: Reply) =
         create Video_Server [S] > new Chan: AV_s >
         new Chan: AV_t > !R.rep(Chan) > delay (2τ_0) >
         ?(5000+2τ,,) Chan [ start() = !Chan.start_ack() >
            ?(40-τ,,) Chan [stop = Stop[Chan]] > !Chan.video() >
            Period_AV[Chan]
         ]
      ] > 0
in
   Video_Server [S]
```

6.5 Conclusion

In this chapter we have reviewed several aspects related to pratical applications of our work. We first tried to root our results deeply into the model-theoretical aspects of reactive system verification. We hope that resulted from this comparison a thoroughly jaggeed image of our motivations and achievements. We then provided a definition for a more tractable, programming-oriented language, that may help introducing our work to persons that have only limited knowledge of the π-calculus and related formalisms. Finally, we showed the direct correspondance between the concepts introduced in $\mathcal{A}$rtOC programs and UML-RT specifications. This should furthermore get our calculi some attention from the lively UML community.

Chapter 7

Conclusion

We have presented a way of reasoning compositionally about real-time, object oriented specifications. To do so, we have extended the π-calculus by introducing clocks that take values over the real numbers. We used transition systems to provide a semantics to processes. The obtained language is extremely expressive, since the transition systems they yield are infinitely branching, and may reach an uncountable number of processes.

We have therefore adapted standard techniques for reasoning about the involved source of infinity in the models. The region graph technique has been used to alleviate the problems caused to the density of the time domain. The symbolic name equality sets and the notion of symbolic bisimulation have been employed to cope with the name-related issues.

We have then been able to analyze processes, in two different aspects. We first provided an axiomatization for timed late bisimilarity over finite terms. Afterwards we allowed to statically ensure the absence of deadlocks and of zeno executions for a subset of infinite-state processes. The static analysis method we provided is based on the fact that the user provides finite-state abstractions of the behavior processes may adopt when using the typed names. Having a strong-enough restriction of name-passing is also a condition of applicability of our method.

Along those developments, the notion of contract has appeared to be essential. We established a parallel with the founding principles of deontic logic, a modal logic where the gap between the factual world and the ideal world is taken into account. This leads to the central notion of directed obligation from a bearer to a counterpart. By allowing processes to explicitly put obligations on their environment, we have reached a strictly higher degree of flexibility and expressiveness than all the other (timed) process algebras that we know. We have shown afterwards that compositional, circular reasoning can be achieved by checking that obligations exhibited in distinct parts of the system are not in contradiction. In this respect, our specifications are assume/guarantee specifications.

We finally showed that real-world applications are available at hand for the proposed notation and verification method. In the world of real-time operating systems and middleware, terms of our algebra can be used as model components to predict, control and regulate application behavior. Contracts passed among components can be used to perform access control and predict resource usage and schedulability.

Well-typed but ill-behaved processes can be freely eliminated by the execution platform by considering that such processes do not respect the contract passed with the platform, to respect the behavior described by their model. We also showed how our notation can be integrated into UML-RT-based developments, yielding the benefits of formal analysis in a place where much is needed.

As future works we may essentially consider further studies of the properties undergone by our models with explicit obligations. This way could indeed reveal itself very fruitful because it provides a very convenient way of specifying configurations of autonomous components. That study would certainly lead to consider finite-state model, where many properties become decidable while they where not in our setting.

The converse angle of attack will be to improve the static analysis method, and to further study the properties of bisimilarity as we defined it for real-time mobile objects. Right now, we indeed have only very low insights on the consequences produced by our definition. A particular point of interest is the subtyping relation defined in Chapter 5. It is quite unsatisfactory in the present state, and one should look for improvements. Finding some kind of elegant formulation for it in terms of games or simulation appears as essential, while we only provided an axiomatic, ad-hoc definition.

Finally, we could also consider many alternative methods that rely for example on true concurrency semantics, categorical semantics, rewriting logic, co-algebraic specifications, etc. Many works dealing with time-related computations have been devised in those contexts, and we can only expect to obtain truck-loads of new solution-generating ideas, and as many new question-raising problems.

Appendix A

Proofs

Proof of Theorem 4.2.1 (Soundness of the abstract semantics).

In each case, by routine induction on the structure of the proof. We start by treating time-passing transitions:

$$t\,\zeta \xrightarrow{\mathcal{S}} t'\,\zeta' \Rightarrow (\forall \rho, \delta.\ (\rho \vDash \zeta \wedge \rho^{+\delta} \vDash ((\zeta^{\nearrow} \backslash \zeta^{=\swarrow}) \wedge \zeta'^{\swarrow})) \Rightarrow t\rho \xrightarrow{\delta\mathcal{S}} t'\rho^{+\delta})\ .$$

We reason about the last rule applied when inferring the symbolic transition, and we show that we can build a concrete proof tree matching the symbolic one. We have to consider the rules Conv, Id, and Reset, as well as all the rules from Tables 4.3 and 4.4.

Axioms Tsel and Tidl are easily matched by tnsel. Suppose indeed

$$([\sigma, \upsilon]\pi.\, t)\,\zeta \xrightarrow{\langle\rangle} ([\sigma, \upsilon]\pi.\, t)(\sigma \wedge \zeta^{\nearrow})^{=\swarrow}$$

by Tsel, with $\zeta' \triangleq (\sigma \wedge \zeta^{\nearrow})^{=\swarrow}$. Suppose also that we have ρ, δ are such that: $\rho \vDash \zeta$ and $\rho^{+\delta} \vDash (\zeta^{\nearrow} \backslash \zeta^{=\swarrow}) \wedge ((\sigma \wedge \zeta^{\nearrow})^{=\swarrow})^{\swarrow}$. We have to prove that for any $\delta' \in (0, \delta)$, $\rho^{+\delta'} \vDash \neg\,\sigma$, hence allowing us to apply Tnsel. From the antecedent of Tsel, $((\zeta \wedge \sigma^{\nearrow}) \Leftrightarrow \bot)$ implies that $\rho \vDash \neg\sigma$, since $\rho \vDash \zeta$. On the other side, we have trivially $(\zeta^{\nearrow} \backslash \zeta^{=\swarrow}) \wedge ((\sigma \wedge \zeta^{\nearrow})^{=\swarrow})^{\swarrow} \Rightarrow (\sigma^{=\swarrow})^{\swarrow}$, and therefore $\rho^{+\delta'} \vDash \neg\sigma$, because $\delta' < \delta$ and $((\sigma^{=\swarrow})^{\swarrow} \wedge \sigma) \Leftrightarrow \sigma^{=\swarrow}$. Hence we obtain the sought concrete transition.

For Tidl the reasoning is similar: suppose that

$$([\sigma, \upsilon]\pi.\, t)\,\zeta \xrightarrow{\langle\rangle} ([\sigma, \upsilon]\pi.\, t)\zeta^{ceil}$$

and take ρ, δ so that $\rho \vDash \zeta$ and $\rho^{+\delta} \vDash (\zeta^{\nearrow} \backslash \zeta^{=\swarrow}) \wedge \zeta^{ceil\swarrow}$. The antecedent of Tidl gives us $(((\zeta^{\nearrow} \backslash \zeta^{=\swarrow}) \wedge \sigma^{\swarrow}) \Leftrightarrow \bot)$, which implies that the only values of ρ that may satisfy both ζ and σ are on the border $\zeta^{=\swarrow}$ (either $\zeta^{\nearrow} \Rightarrow \neg\sigma$, or $\zeta^{=\swarrow} \Rightarrow \sigma$ and $(\zeta^{\nearrow} \backslash \zeta^{=\swarrow}) \Rightarrow \neg\sigma$). Since this border is excluded from $(\zeta^{\nearrow} \backslash \zeta^{=\swarrow}) \wedge \zeta^{ceil\swarrow}$ to which $\rho^{+\delta}$ belongs, we deduce indeed that $\rho^{+\delta'} \vDash \neg\sigma$ for any $\delta' \in (0, \delta)$. We hence may apply tnsel to match Tidl transitions.

A deduction using axiom Turg can be proved to correspond to a concrete deduction using axiom tsel by using a slight variation of the proof given above for axiom Tsel. We have $\zeta' \triangleq (\upsilon \wedge \zeta^{\nearrow})^{=\swarrow}$ and we take δ such that $\rho^{+\delta} \vDash (\zeta^{\nearrow} \backslash \zeta^{=\swarrow}) \wedge ((\upsilon \wedge \zeta^{\nearrow})^{=\swarrow})^{\swarrow}$. From this and the antecedent of Turg, we easily get $\rho^{+\delta} \vDash \zeta^{\nearrow} \wedge \upsilon^{=\swarrow}$. Therefore $\rho^{+\delta}$ is

on the lower border verifying urgency condition υ. Since $\rho \vDash \sigma$ from the antecedent of TURG and since $\sigma \Rightarrow \upsilon$, we obtain that $\rho^{+\delta'} \vDash \sigma \wedge \neg\upsilon$ for any $\delta' \in [0, \delta)$. Hence TSEL applies and produces a corresponding concrete transition.

A deduction using symbolic axiom TELA can also be matched by a concrete deduction using axiom TSEL. This time, we have $\zeta' \triangleq (\sigma \wedge \zeta^{\nearrow})^{\nearrow^{=}}$ and thus $\rho^{+\delta} \vDash (\zeta^{\nearrow} \backslash \zeta^{=\swarrow}) \wedge ((\sigma \wedge \zeta^{\nearrow})^{\nearrow^{=}})^{\swarrow}$. By the antecedent in TELA, we have $(\upsilon \wedge \zeta^{\nearrow}) \Leftrightarrow \bot$, implying that $\rho^{+\delta} \vdash \neg\upsilon$ whatever δ. From the antecedent we also have $\sigma^{\nearrow} \not\Leftrightarrow \sigma$, implying that σ is not umbounded, and that consequently $\sigma^{\nearrow^{=}}$ exists. Since trivially $\rho^{+\delta} \vDash \zeta^{\nearrow} \wedge \sigma \wedge (\sigma^{\nearrow^{=}})^{\swarrow}$, for all $\delta' \in [0, \delta)$, $\rho^{\delta'} \vDash \sigma \backslash \sigma^{\nearrow^{=}}$. Thus $\rho^{+\delta'} \vDash \sigma \wedge \neg\upsilon$, and TSEL can be applied.

For TUNBS, the destination zone is ζ^{ceil}, and we have $\rho^{+\delta} \vDash (\zeta^{\nearrow} \backslash \zeta^{=\swarrow}) \wedge \zeta^{ceil\swarrow}$. From the antecedent of the rule, we get as in the previous case that the intersection between $\zeta^{\nearrow}$ and υ is empty, hence provinding $\rho^{+\delta} \vDash \neg\upsilon$ for any δ. From the two other clauses ($\zeta \Rightarrow \sigma$ and $\sigma^{\nearrow} \Leftrightarrow \sigma$), we get that σ is unbounded, and therefore $\rho^{+\delta} \vDash \sigma$ for any δ. Thus we infer that $\rho^{+\delta'} \vDash \sigma \wedge \neg\upsilon$ for any $\delta' \in [0, \delta)$, and TSEL can be applied.

The case of TUNBU is very-similar to TUNBS, but the concrete matching rule is TURG. From the antecedent of the rule we have $\rho \vDash \upsilon$ (since $\zeta \Rightarrow \upsilon$) and $\rho^{+\delta} \vDash \upsilon$ for any δ (since $\upsilon^{\nearrow} \Leftrightarrow \upsilon$). Thus $\rho^{+\delta'} \vDash \upsilon$ for any $\delta' \in [0, \delta]$, and TURG may apply.

For TMISS, we can invoke TMISS: if we have the following symbolic transition

$$([\sigma, \upsilon]\pi^{\star}.\, t)\, \zeta \xrightarrow{\langle s(\pi^{\star}):\emptyset\rangle} Error((\upsilon \wedge \zeta^{\nearrow})^{\nearrow^{=}})$$

then we can show that all the instants belonging to the destination zone are the ealiest possible when an error is reached. Indeed take ρ, δ so that $\rho \vDash \zeta$ and $\rho^{+\delta} \vDash (\zeta^{\nearrow} \backslash \zeta^{=\swarrow}) \wedge ((\upsilon \wedge \zeta^{\nearrow})^{\nearrow^{=}})^{\swarrow}$. From the antecedent of TMISS, we get the υ is not unbounded (by $\upsilon^{\nearrow} \not\Leftrightarrow \upsilon$), which has for consequence that $\upsilon^{\nearrow^{=}}$ defined, and by the form of urgency criteria, $(\upsilon^{\nearrow^{=}} \wedge \upsilon) \Leftrightarrow \bot$ (υ must have an upper constaint of the form $x < p$, which is replaced by a constraint $x = p$ in $\upsilon^{\nearrow^{=}}$). Therefore, as $\rho^{+\delta} \vDash (\zeta^{\nearrow} \backslash \zeta^{=\swarrow}) \wedge ((\upsilon \wedge \zeta^{\nearrow})^{\nearrow^{=}})^{\swarrow}$ implies $\rho^{+\delta} \vDash \upsilon^{\nearrow^{=}}$, we have $\rho^{+\delta} \vDash \neg\upsilon$. We are left to show $\forall \delta' \in [\,0, \delta)\ \rho^{+\delta'} \vDash \upsilon$. It is easy, since by the antecedent of TMISS, $\zeta \Rightarrow \upsilon$, and that all instants before δ must satisfy $(\upsilon \wedge \zeta^{\nearrow})^{\nearrow^{=}}$. So TMISS can apply.

The concrete axiom TERR is powerful enough to cope with symbolic axioms TERR and TNERR. The proofs are similar to the TMISS and TIDL cases, respectively. We are left with only one axiom, TREQ, which is matched by TURG, and for which the treatment is similar to the one we are now going to engage for rule TREG. The main difference between the two is that we must involve an induction hypothesis to properly negociate TREG.

Let us therefore suppose that the following transition has been proved:

$$([\sigma, \upsilon]\pi.\, t)\, \zeta \xrightarrow{s} ([\sigma, \upsilon]\pi.\, t)\, \zeta'\,.$$

The proof of such a transition is mandatorily an axiom among TSEL, TURG, TELA, TIDL, TUNBS, TUNBU or TREQ, followed by zero or more (but finitely many) applications of TREG. The transition is in any case matched by one of the concrete axioms TNSEL, TSEL, or TURG, according to the symbolic axiom used at the root of the deduction tree. When only a symbolic axiom is used, the existence of a concrete deduction matching TREG is proved by using the interval-trajectory property of

our models, as defined in Section 3.8. Having effectively found a matching concrete transition $t\rho \xrightarrow{\delta\,\mathcal{S}} t'\rho^{+\delta}$ by using one of the proof cases for symbolic axioms detailed above, we have proved that for any $\delta' \in (0,\delta)$, $\rho^{+\delta'}$ satisfies a certain condition. This remains obviously true for any $\delta'' \in [0,\delta']$. The same concrete axiom may hence be applied to infer any δ'-transition. The rule TREG states that any point in time between the current zone and the destination zone is reachable by a time-passing transition. That is, the destination zone ξ is included in $(\zeta^{\nearrow}\backslash\zeta^{=\swarrow}) \wedge \zeta'^{\swarrow}$, and for any clock valuation κ such that $\kappa \models \xi$, there is a δ'' such that $\delta'' \in (0,\delta)$ and $\kappa = \rho^{+\delta''}$. This reasoning can be repeated for any number of successive applications of rule TREG.

Deductions employing TSUM and TPAR can be matched respectively by using TSUM and TPAR. Both proofs are nearly identical, we give only the one for TPAR.

Suppose, as we said, that:

$$t\,\zeta^{\downarrow x} \xrightarrow{\mathcal{S}_t} t'\,\zeta' \Rightarrow (\forall \rho_t, \delta_t.\ (\rho_t \models \zeta^{\downarrow x} \wedge \rho_t^{+\delta_t} \models ((\zeta^{\downarrow x\nearrow}\backslash\zeta^{\downarrow x=\swarrow}) \wedge \zeta'^{\swarrow})) \Rightarrow t\rho_t \xrightarrow{\delta_t\,\mathcal{S}_t} t'\rho_t^{+\delta_t})$$

and

$$u\,\xi^{\downarrow x} \xrightarrow{\mathcal{S}_u} u'\,\xi' \Rightarrow (\forall \rho_u, \delta_u.\ (\rho_u \models \xi^{\downarrow x} \wedge \rho_u^{+\delta_u} \models ((\xi^{\downarrow x\nearrow}\backslash\xi^{\downarrow x=\swarrow}) \wedge \xi'^{\swarrow})) \Rightarrow u\rho_u \xrightarrow{\delta_u\,\mathcal{S}_u} u'\rho_u^{+\delta_u})$$

and suppose that the rule TPAR applies. From TPAR we infer

$$(t \mid u)\,(\zeta \wedge \xi) \xrightarrow{\mathcal{S}_t,\mathcal{S}_u} (t' \mid u')\,(\zeta' \wedge \xi')^{\backslash x}\ .$$

We have to prove that

$$\begin{array}{l}(t \mid u)\,(\zeta \wedge \xi) \xrightarrow{\mathcal{S}_t,\mathcal{S}_u} (t' \mid u')\,(\zeta' \wedge \xi')^{\backslash x} \\ \Rightarrow (\forall \rho, \delta.\ \ (\rho \models (\zeta \wedge \xi) \wedge \rho^{+\delta} \models ((((\zeta \wedge \xi)^{\nearrow}\backslash(\zeta \wedge \xi)^{=\swarrow}) \wedge ((\zeta' \wedge \xi')^{\backslash x})^{\swarrow}) \\ \qquad\qquad \Rightarrow (t \mid u)\rho \xrightarrow{\delta\,\mathcal{S}_t,\mathcal{S}_u} (t' \mid u')\rho^{+\delta}))\end{array}$$

This means that we must find a way to make rule TPAR apply; to do this we must show that the antecedent of TPAR is verified, yielding one concrete transition for each process t and u. Suppose that $\rho \models (\zeta \wedge \xi)$. We call ρ_t and ρ_u the clock valuations such that for some $x \notin dom(\zeta) \cup dom(\xi)$, $dom(\rho_t) = clocks(\zeta) \cup \{x\}$, $dom(\rho_u) = clocks(\xi) \cup \{x\}$, $\forall y \in clocks(\zeta).\rho(y) = \rho_t(y)$, $\forall z \in clocks(\xi).\rho(z) = \rho_u(z)$, and $\rho_t(x) = \rho_u(x) = 0$. This yields $\rho = (\rho_t\rho_u)^{\backslash x}$, $\rho_t \models \zeta^{\downarrow x}$, and $\rho_u \models \xi^{\downarrow x}$. We then show that for all δ such that $\rho^{+\delta} \models ((((\zeta \wedge \xi)^{\nearrow}\backslash(\zeta \wedge \xi)^{=\swarrow}) \wedge ((\zeta' \wedge \xi')^{\backslash x})^{\swarrow})$, we have $\rho_t^{+\delta} \models ((\zeta^{\downarrow x\nearrow}\backslash\zeta^{\downarrow x=\swarrow}) \wedge \zeta'^{\swarrow})$ and $\rho_u^{+\delta} \models ((\xi^{\downarrow x\nearrow}\backslash\xi^{\downarrow x=\swarrow}) \wedge \xi'^{\swarrow})$. this will provide us the wanted result, by application of the induction hypothesis.

All the instants reachable by letting a time δ pass from some instant verifying constaint ζ define the zone $\zeta^{\nearrow}$. Hence, $\rho_t \models \zeta^{\downarrow x}$ yields trivially $\rho_t^{+\delta} \models \zeta^{\downarrow x\nearrow}$. From the obligation that $\rho^{+\delta} \models (\zeta \wedge \xi)^{\nearrow}\backslash(\zeta \wedge \xi)^{=\swarrow}$, we further infer that $\delta > 0$, the lower border of $(\zeta \wedge \xi)$ being excluded. This allows us to exclude also the lower border of $\zeta^{\downarrow x\nearrow}$, and to infer $\rho_t^{+\delta} \models \zeta^{\downarrow x\nearrow}\backslash\zeta^{\downarrow x=\swarrow}$. We now show that $\rho^{+\delta_t} \models (\zeta' \wedge \xi')^{\backslash x}$ implies $\rho_t^{+\delta_t} \models \zeta'$. Indeed, the constraints ζ' and ξ' have been obtained by taking the upward opening of $\zeta^{\downarrow x}$ and $\xi^{\downarrow x}$, respectively. Considering only ζ (the reasoning is similar for ξ) this implies, for any clock $z \in clocks(\zeta)$, that $\zeta' \Rightarrow z - x \sharp r$ for some r. As furthermore the clock x must not appear in the term t (by the side-condition on rule TPAR), then any constraint on x in ζ' such that $x \sharp p$ induces the existence of

a clock y and a constant q such that $\zeta' \Rightarrow (y \sharp q \wedge y - x \sharp q - p)$. Reasoning by absurdity, suppose now that $\rho^{+\delta_t} \vDash (\zeta' \wedge \xi')^{\backslash x}$ for some δ_t, while $\rho_t^{+\delta_t} \nvDash \zeta'$. The non-satisfaction of ζ' by ρ_t is either due to $(\rho_t^{+\delta_t})^{/y} \nvDash \zeta'^{/y}$ for some $y \in clocks(\zeta)$ or to $(\rho_t^{+\delta_t})^{/x} \nvDash \zeta'^{/x}$. In the first case, we trivially obtain a contradiction, since necessarily $\rho^{+\delta_t} \nvDash \zeta'$, which implies $\rho^{+\delta_t} \nvDash (\zeta' \wedge \xi')^{\backslash x}$. In the second case, we are given by the form of ζ' the existence of $y \in clocks(\zeta)$ such that $\zeta' \Rightarrow y \sharp q$ and $(\rho^{+\delta_t})^{/y} \nvDash \zeta'^{/y}$. This closes the absurdity reasoning, proving that $\rho^{+\delta_t} \vDash (\zeta' \wedge \xi')^{\backslash x}$ implies $\rho_t^{+\delta_t} \vDash \zeta'$. By property of the downward closing operator, we get $\rho^{+\delta_t} \vDash ((\zeta' \wedge \xi')^{\backslash x})^{\downarrow}$ implies $\rho^{+\delta_t} \vDash \zeta'^{\downarrow}$, and by simple conjunction $\rho^{+\delta_t} \vDash (((\zeta \wedge \xi)^{\uparrow} \backslash (\zeta \wedge \xi)^{=\downarrow}) \wedge ((\zeta' \wedge \xi')^{\backslash x})^{\downarrow})$ implies $\rho_t^{+\delta_t} \vDash ((\zeta^{\downarrow x\uparrow} \backslash \zeta^{\downarrow x=\downarrow}) \wedge \zeta'^{\downarrow})$. The same reasoning can be repeated for ρ_u, ξ, and δ_u. Hence we can use the induction hypothesis and apply rule TPAR to match rule TPAR: we have $(t \mid u)\rho \xrightarrow{\delta\, \mathcal{S}_t, \mathcal{S}_u} (t' \mid u')\rho^{+\delta}$.

Now for untimed transitions we have to prove:

$$\begin{aligned} t\,\zeta \xrightarrow{\pi,\mu} t'\,\zeta' \Rightarrow ((\forall \rho, \alpha.\, ((\mathrm{bn}(\pi) \cap (\mathrm{fn}(t) \cup ports(\alpha))) = \emptyset \wedge \rho \vDash \zeta \wedge \alpha \vDash \mu) \\ \Rightarrow (\exists \rho'.\, \rho' \vDash \zeta' \wedge (t\alpha)\rho \xrightarrow{\pi\alpha} (t'\alpha)\rho')) \wedge \\ (\forall \rho', \alpha.\, ((\mathrm{bn}(\pi) \cap (\mathrm{fn}(t) \cup ports(\alpha))) = \emptyset \wedge \rho' \vDash \zeta' \wedge \alpha \vDash \mu) \\ \Rightarrow (\exists \rho.\, \rho \vDash \zeta \wedge (t\alpha)\rho \xrightarrow{\pi\alpha} (t'\alpha)\rho'))\ . \end{aligned}$$

This property is very similar to the second clause of [Lin94, Lemma 2.12], the only difference being in the need for us to prove the existence of two oppositely-directed functions between the clock valuations of ζ and the ones of ζ'.

We have, as in the timed case, to prove the correspondance for every rule and axiom of Table 4.2 a suitable correspondence in Table 3.2, using the usual induction on the struture of terms.

We start by the only symbolic axiom PRE, that shall be matched by axiom PRE. The capacity of inferring the concrete transition is maintained when renaming free ports of π, t, and t'. Thus the application of α can be done freely in the consequent of the theorem. We now deal with timing aspects. In the case of a port input or clock/port output, matching the symbolic transition is trivial: $\zeta' = reach(\zeta, \pi) = \zeta$, $reach(\rho, \pi) = \rho$, and from the precondition of PRE, $\zeta \Rightarrow \sigma$, implying that for any ρ we have $\rho \vDash \zeta \Rightarrow \rho \vDash \sigma$; this allows the application of PRE. In the case of clock input, the received value x is undetermined. We have $\zeta' = reach(\zeta, \pi) = (\zeta \wedge x \geq 0)$, and $\rho'^{\backslash x} = \rho$; trivially for any ρ such that $\rho \vDash \zeta$, $\rho' \vDash \zeta'$ because ζ' does not constrain the value of x. The existence of a function from each ρ satisfying ζ to ρ' satisfying ζ' and conversely is trivial in the first case; in the second case, the function is injective from ρ to ρ', and surjective in the converse direction (there are infinitely many values for x in ρ' such that $\rho'^{\backslash x} = \rho$).

The proofs for all the remaining rules follow roughly the pattern we just used for PRE. The naming issue for those rules is treated in [Lin03, lemma 2.6, second clause] (the interesting cases being RES, OPEN, and MATCH). We have to deal with timing issues in the cases OPEN, RES, SUM, ID, PAR, MATCH and CONV. In each of those cases, the same origin and destination zones ζ and ζ' appear in the antecedent and the consequent of the rule. The existence of two oppositely-directed functions between their satisfying clock valuations is therefore immediate from the induction hypothesis. The only remaining cases are COM, CLOSE, and RESET, which are trivial. In RESET, the clock reset of x is the only change from ζ to ζ'. The case

for CLOSE is trivial using the induction hypothesis: we have functions between ζ and ζ', and ξ and ξ'. Finding corresponding functions between $\zeta \wedge \xi$ and $\zeta' \wedge \xi'$ amounts to taking the same values on common clocks, and the union of values of reset clocks. Finally, the case for COM solves easily from the previous one, since the only difference is that the value of clock m (if it is a clock) is suppressed in ζ.

□

Proof of Theorem 4.2.1 (Weak Completeness of the abstract semantics).

We have to prove that, to any concrete transition, there corresponds at least one abstract transition. As in the soundness proof, we start by dealing with the timed case. Formally, the clause is written:

$$t\rho \xrightarrow{\delta \mathcal{S}} t'\rho' \Rightarrow (\exists\, \zeta.\ \rho \models \zeta \wedge (\rho^{+\delta} \models \zeta \vee (\exists\, \zeta'.\ \rho^{+\delta} \models \zeta' \wedge t\,\zeta \xrightarrow{\mathcal{S}} t'\,\zeta')))$$

We have to perform a case analysis on rules RESET, ID, and CONV of Table 3.2, and all rules in Table 3.3.

More precisely, we show that the region enclosing ρ, that we shall note ζ_ρ, is always a possible choice respecting the wanted property. Such a region can be obtained easily by, for any clock x, having $\zeta_\rho \Rightarrow (x > \lfloor\rho(x)\rfloor \wedge x < \lfloor\rho(x)\rfloor + 1)$ when $\rho(x) \neq \lfloor\rho(x)\rfloor$ and $x = \rho(x)$ otherwise, and for any other clock y, $\zeta_\rho \Rightarrow (x - y > \lfloor\rho(x) - \rho(y)\rfloor \wedge x - y < \lfloor\rho(x) - \rho(y)\rfloor + 1)$ when $\rho(y) < \rho(x)$, and $\zeta_\rho \Rightarrow x = y$ when $\rho(x) = \rho(y)$. The region ζ_ρ enclosing a given clock valuation is always an equivalence class for the valuations it contains [AD94], meaning that all those valuations satisfy exactly the same constraints.

The axiom TNSEL can be matched by TREG and either TSEL or TIDL, depending on whether $\rho^{+\delta} \models \sigma$ or not. In the first case, we have both $\zeta_\rho \Rightarrow \neg\sigma$ and $\zeta_{\rho^{+\delta}} \Rightarrow \sigma$ from the antecedent of the axiom TNSEL. The antecedent of rule TSEL is then easily verified:

- because a valuation satisfying σ is reachable from ρ we have $((\zeta_\rho \wedge \sigma^{\nearrow}) \Leftrightarrow \bot)$,
- because $\zeta_{\rho^{+\delta}} \Rightarrow \sigma$ and $\zeta_{\rho^{+\delta}}$ is an equivalence class we have $\zeta^{\nearrow\swarrow} \Rightarrow \sigma^{\nearrow\swarrow}$.

Finally, from the fact that $\forall \delta' \in (0, \delta)\ \rho^{+\delta'} \models \neg\,\sigma$, we are assured that the destination region $\zeta_{\rho^{+\delta}}$ is the lower border of the intersection of $\zeta_\rho^{\nearrow}$ and σ: $\zeta_{\rho^{+\delta}} \Leftrightarrow (\sigma \wedge \zeta_\rho^{\nearrow})^{=\swarrow}$.

The case where $\rho^{+\delta} \models \neg\,\sigma$, has to be divided again in two subcases. The first one treats cases where time may elapse to make σ eventually *true*, but has not elapsed enough yet. Then, an application of TREG after TSEL provides the wanted symbolic transition. The applicability of TREG is guaranteed by the fact that $\delta > 0$ and $\delta' > 0$. In the other case, where σ will never become *true* whatever amount of time elapses, TIDL must be employed first, potentially followed by TREG. Because in this case $\neg\,\sigma$ is verified at all points between ζ_ρ and $\zeta_{\rho^{+\delta}}$, the intersection of this zone with $\sigma^{\swarrow}$ is mandatorily empty, allowing the application of TIDL. The reached zone is then ζ_ρ^{ceil}; the region $\zeta_{\rho^{+\delta}}$ is then logically contained in $(\zeta_\rho^{\nearrow} \backslash \zeta_\rho^{=\swarrow}) \wedge (\zeta_\rho^{ceil})^{\swarrow}$, allowing the application of TREG.

The axiom TSEL shall undergo a similar treatment, using a combination of TREG with either TURG, TELA, or TUNBS. The cases to be covered are:

- the ones where either the currently allowed prefix becomes urgent or will eventually become urgent but for now remains selected (covered by using TURG and TREG),

- the case where the selection constraint will eventually become $false$ again (covered by TELA and TREG),

- the case where the selection constraint is unbounded (covered by TUNBS and TREG).

We do not treat those cases in detail, as they are very similar to the treatments applied to TNSEL.

The treatment for axiom TURG is more simple: since time may not elapse to that an urgency condition υ becomes false again, time may only pass withing the limits of a bounded constraint (by using TREQ), or it may reach the upper limit of an unbounded constraint (by using TUNBU and TREG). The axiom TMISS is matched trivially by TMISS, the only reachable region being the lower border of the zone where υ become $false$. The axiom TERR is matched by TERR and TNERR, that correspond respectively to the two clauses of the antecedent of TERR. While TRES is matched by TRES, we are only left with the two cases for parallel and choice connectors, that demand the use of the traditional structural induction method. We treat only the parallel case.

Suppose that a time progression is decided by using rule TPAR; then, TPAR can be used to find a matching abstract transition. By the induction hypothesis, we have indeed two transitions $t\zeta_{\rho^{\downarrow x}} \xrightarrow{\delta \mathcal{S}} t'\zeta_{\rho^{\downarrow x}+\delta}$ and $u\xi_{\kappa^{\downarrow x}} \xrightarrow{\delta \mathcal{S}'} u'\xi_{\kappa^{\downarrow x}+\delta}$. Applying TPAR can be done easily, because the side-condition of TPAR implies that the one for TPAR is respected: while the constraints on the ready-sets are directly carried on from rule to rule, the fact that $\rho^{\downarrow x}\kappa^{\downarrow x}$ is defined implies that $(\zeta_{\rho^{\downarrow x}} \wedge \xi_{\kappa^{\downarrow x}}) \not\Leftrightarrow \bot$, $(\zeta_{\rho^{\downarrow x}+\delta} \wedge \xi_{\kappa^{\downarrow x}+\delta}) \not\Leftrightarrow \bot$, and $\zeta_{\rho^{\downarrow x}+\delta}$ and $\xi_{\kappa^{\downarrow x}+\delta}$ agree on the possible values of all the clocks they have in common.

We now address the discrete case. The proofs relies on Hennessy and Lin's work [Lin94], but it has a slight difference that clocks may be updated (*i.e.* reset) during discrete transitions. In spite of this difference, the proof is very-similar to the above-mentioned work, and it goes as usual by induction on the structure of terms, reasoning on the rule that is applied last. We hence consider the rules and axioms of Tables 4.2 and 3.2.

The only axiom is PRE, that is easily matched by PRE. Take indeed the region (*i.e.* the smallest zone) ζ such that $\rho \vDash \zeta$. Since $\rho \vDash \sigma$ and ζ is an equivalence class for all $t\rho$ belonging to it, we have $\zeta \Rightarrow \sigma$. By the definition of $reach(\rho, \pi\alpha)$ and $reach(\zeta, \pi)$, we have mandatorily $\rho' \in reach(\rho, \pi\alpha) \Rightarrow \rho' \vDash reach(\zeta, \pi)$. Therefore the precondition of PRE is satisfied, and we obtain the sought symbolic transition.

The other cases use the induction hypothesis on structure of terms, sometimes in a straightforward way. This is the case for SUM matched by SUM, PAR matched by PAR, RESET matched by RESET, and ID matched by ID. In all those cases, the side conditions of the concrete and symbolic rules are the same, and the distinction set μ on the antecedent and consequent abstract transitions remains unchanged; the

proof is hence trivial. We have now only the cases OPEN, CLOSE, COM, RES and CONV that remain to be treated.

The case of OPEN is resolved by employing OPEN. By using the induction hypothesis on $t\rho \xrightarrow{\overline{a}b} t'\rho'$, we have the existence of a name constraint μ such that $t\,\zeta \xrightarrow{\overline{a}b,\mu} t'\,\zeta'$. To apply OPEN, we then have to prove that $b \notin (\mathrm{ports}(\mu) \cup \{a\})$. When inferring the wanted abstract transition of process $t\,\zeta$, the only way to introduce a constraint on name b is to use rule MATCH; the term t has then the form $[a = b]\overline{a}b.t'$. Reasoning *ad absurdum*, we conclude the impossibility of inferring the supposed concrete transition for $(\nu b\ t)\rho$ using OPEN, and therefore $b \notin \mathrm{ports}(\mu)$. As furthermore from the application of OPEN we get $\alpha(b) \neq \alpha(a)$, and because α is a function (each name has a unique substitution), we can only have $a \neq b$. The side-condition of OPEN is hence shown fulfilled. The timing aspects of the proof are dealt with in a very simple way, by taking the smallest zone containing ρ, and taking for ζ' the same zone with a set value of 0 for all the clocks reset during the transition.

The case for CLOSE is solved by using CLOSE. The induction hypothesis gives us the existence of constraints μ and η, which, by conjucting θ with them, ensures the existence of a proper name constraint on the transition. The side conditions of the rules only bear on timing aspects; the one for CLOSE is easily satisfied by taking for ζ and ξ the smallest regions respectively satisfying ρ and κ. Because $\rho\kappa$ and $\rho'\kappa'$ are defined, it means that in both couples the constraints agree on the clocks they have in common, and therefore the intersection of ζ with ξ (resp. of ζ' with ξ') are non-empty. The case for COM is very-similar to the previous one (with a match by COM), except that the value of a clock m must be suppressed in one of the environments to simulate its transmission.

The last case, for RES, is dealt with using RES. From $\alpha(a) \notin n(\pi\alpha)$, we get that $a \notin n(\pi)$. Then $a \notin n(\mu)$ is obtained *ad absurdum* as in the OPEN case, because if μ contains a constraint $a = b$, it means that the rule MATCH has been employed with a and b syntactically different, while a concrete transition can not be obtained with the same term (by congruence, $[a = b]t \equiv 0$ whatever t).

□

Proof of Theorem 4.2.3 (Relating Concrete and Symbolic Bisimulations).

To prove the implication part of the theorem, we exhibit as usual a concrete bisimulation from the symbolic one, for any $\alpha \vDash \mu$, $\rho \vDash (\zeta^{\downarrow x\uparrow} \wedge \zeta)$ and $\kappa \vDash (\xi^{\downarrow x\uparrow} \wedge \xi)$ with $x \notin (clocks(\zeta) \cup clocks(\xi))$ and $\rho(x) = \kappa(x)$.Assume there is a symbolic timed late bisimulation R^μ between processes $t\,\zeta$ and $u\xi$. We shall show that

$$S = \{((t\alpha)\rho^{\backslash x}, (u\alpha)\kappa^{\backslash x}) \mid \exists \zeta, \xi \text{ so that } (t\,\zeta, u\xi) \in R^\mu, x \notin (clocks(\zeta) \cup clocks(\xi)), \alpha \vDash \mu, \rho \vDash (\zeta^{\downarrow x\uparrow} \wedge \zeta), \kappa \vDash (\xi^{\downarrow x\uparrow} \wedge \xi), \rho(x) = \kappa(x)\}$$

is a (concrete) bisimulation. This goes, for each possible concrete transition of a left-most term in a couple of S, by using the weak completeness theorem to infer a matching symbolic transition, and then, because R^μ is a bisimulation, proving the existence of a right-most concrete term matching the transition with the soundness theorem. We purposedly ignore naming issues, and thus the renaming α, since this

renaming can not interfere with free (or bound) clock names, since those names are always convertible.

Suppose that, for $(t\rho^{\backslash x}, u\kappa^{\backslash x}) \in S$, we have $t\rho^{\backslash x} \xrightarrow{\delta\mathcal{S}} t'\rho'$; then the weak completeness theorem gives us, for each $\rho^{\backslash x}$, the existence of a constraint ζ' such that $\rho^{\backslash x} \vDash \zeta' \wedge ((\rho^{\backslash x})^{+\delta} \vDash \zeta' \vee (\exists\, \zeta''.\ (\rho^{\backslash x})^{+\delta} \vDash \zeta'' \wedge t\,\zeta' \xrightarrow{\mathcal{S}} t'\,\zeta''))$. We have furthermore $\rho \vDash (\zeta^{\downarrow x\uparrow} \wedge \zeta)$, which implies $\rho^{\backslash x} \vDash (\zeta^{\downarrow x\uparrow} \wedge \zeta)^{\backslash x}$, while trivially $(\zeta^{\downarrow x\uparrow} \wedge \zeta)^{\backslash x} \Leftrightarrow \zeta$. We obtain, by $\rho^{\backslash x} \vDash \zeta$ and $\rho^{\backslash x} \vDash \zeta'$, that $\rho \vDash (\zeta \wedge \zeta')$ and $(\zeta \wedge \zeta') \not\Leftrightarrow \bot$. There are now two cases to consider: either there exists ζ_i member of a slice partition $\zeta_1, \cdots, \zeta_k$ of $(\zeta^{\downarrow x\uparrow} \wedge \zeta)$ having the properties given by Definition 4.2.3 and such that $\zeta' \Rightarrow \zeta_i$, or there is no such ζ_i. In the first case, we can take ζ' to be equivalent to any suitable $\zeta_i^{\backslash x}$ since, by the definition of a slice partition (Definition 4.2.1), this $\zeta_i^{\backslash x}$ also has the property required above by the weak completeness theorem. In the latter case, we have a ζ' that spans over many members of $\zeta_1, \cdots, \zeta_k$. We can then choose a $\zeta_i^{\backslash x}$ so that $\rho \vDash \zeta_i$, that has again the same time-passing properties as ζ' regarding the weak completeness theorem. Now, by the downward closure property of timed late bisimulation (and thus of R^μ) over the $\zeta_1, \cdots, \zeta_k$, we deduce that the term $t\,\zeta_i^{\backslash x}$ is bisimilar to another symbolic term $u\,\xi_i^{\backslash x}$ by R^μ.

We are now interested only in the subset of clock valuations that share the same property as the initially chosen ρ: the set of valuations (that we all call ρ for convenience) verifying $\rho \vDash \zeta_i$. Whenever $t\,\zeta_i^{\backslash x}$ produces a transition leading to some $t\,\zeta''$ in order to match the transition of $t\rho^{\backslash x}$ (this is the case when $(\rho^{\backslash x})^{+\delta} \not\vDash \zeta_i^{\backslash x}$), then the symbolic process $u\,\xi_i^{\backslash x}$, bisimilar to $t\,\zeta_i^{\backslash x}$ by R^μ, produces the same symbolic transition, leading to $u\xi'$. We can then apply the soundness theorem to that symbolic transition, obtaining that for any $\kappa^{\backslash x} \vDash \xi_i^{\backslash x}$, there is a concrete transition from $u\kappa^{\backslash x}$ to $u(\kappa^{\backslash x})^{+\delta'}$ for any δ' such that $(\kappa^{\backslash x})^{+\delta'} \vDash \xi'$; let us call Δ the set of such δ'. The fact that $\delta \in \Delta$ is given by the first clause of the bisimulation definition: δ is such that $\rho^{\backslash x} \vDash \zeta_i^{\backslash x}$ implies $(\rho^{\backslash x})^{+\delta} \vDash \zeta''$; besides, $(t\,\zeta_i^{\backslash x}, u\,\xi_i^{\backslash x}) \in R^\mu$ implies that ζ'' and ξ'' are obtained from two other zones ζ''' and ξ''' by suppression of the constraints on some new clock y. Finally, the first clause of Definition 4.2.3 also enforces that ζ''' and ξ''' have the same constant width, and that $\zeta'''^{/y} \Leftrightarrow \xi'''^{/y}$. This closes the case where a symbolic transition is produced by $t\,\zeta_i$ to match the concrete δ-transition.

If no symbolic transition is produced, it means that $(\rho^{\backslash x})^{+\delta} \vDash \zeta_i^{\backslash x}$. The existence of a matching concrete transition is given by the constraint over the clock x in the definition of symbolic timed late bisimulation. Precisely, to show that $(t\,\zeta_i^{\backslash x}, u\,\xi_i^{\backslash x}) \in R^\mu$, we have used the fact that $(t\,\zeta, u\xi) \in R^\mu$ and that ζ_i belongs to some slice partition of ζ. Since the definition imposes also that ζ_i and ξ_i have the same constant width and $\zeta_i^{/x} \Leftrightarrow \xi_i^{/x}$, any time progression δ such that $(\rho^{\backslash x})^{+\delta} \vDash \zeta_i^{\backslash x}$ can be trivially matched by $\xi_i^{\backslash x}$. This closes the timed case.

The discrete transition case goes along the same line as the timed case. Suppose that, for $((t\alpha)\rho^{\backslash x}, (u\alpha)\kappa^{\backslash x}) \in S$, we have $(t\alpha)\rho^{\backslash x} \xrightarrow{\pi\alpha} t'\rho'$. We have then to show that $(u\alpha)\kappa$ is able to produce a matching transition whenever $bn(\pi\alpha) \cap (\mathrm{fn}(t\alpha) \cup \mathrm{fn}(u\alpha)) = \emptyset$ (by the definition of timed late bisimulation, we have otherwise nothing to do). We are able to apply the clause on discrete transitions of the weak completeness theorem: $bn(\pi) \cap (\mathrm{fn}(t\alpha) \cup \mathrm{fn}(u\alpha)) = \emptyset$ implies $bn(\pi) \cap \mathrm{fn}(t\alpha) = \emptyset$ and therefore there exists μ' such that $\alpha \vDash \mu'$ and we have an abstract transition for some $t\,\zeta''$

wearing π, μ' as a label. Since $\alpha \vDash \mu'$ and $\alpha \vDash \mu$, we infer that $(\mu \wedge \mu') \not\Leftrightarrow \bot$, because there can be no a and b such that $\mu \Rightarrow (a = b)$ while $\mu' \Rightarrow (a \neq b)$ (or vice-versa), as otherwise we would have both $\alpha(a) = \alpha(b)$ and $\alpha(a) \neq \alpha(b)$. Hence the set $MCE_{\mathrm{fn}(t)\cup\mathrm{fn}(u)}(\mu \wedge \mu')$ is not empty, and by $(t\,\zeta, u\,\xi) \in R^{\mu}$ we obtain a matching transition with label π', θ' for a process $u\xi''$ respecting the second clause of Definition 4.2.3 regarding $t\,(\zeta \wedge \zeta'')$ whatever ζ'' and θ' (with $\theta \Rightarrow \theta'$) can be. Proving that ζ and ζ'' have a non-empty intersection is done as before, resulting again in the existence of a matching transition, this time by the downward partitioning property of zones that are member of a slice partition.

We finally apply the soundness theorem to the transition of $u\,\xi''$. This yields for any $\kappa \vDash (\xi'')$ the existence of a clock valuation κ' such that there is a concrete transition $(u\alpha)\kappa \xrightarrow{\pi'\alpha} (u'\alpha)\kappa'$. This in turn yields the existence of a matching transition for any $t\rho$ with $\rho \vDash (\zeta \wedge \zeta'')^{\backslash x}$. The converse case is also handled by the soundness theorem, ensuring the existence of a transition for any ρ' such that $(t\alpha)\rho^{\backslash x} \xrightarrow{\pi\alpha} t'\rho'$. The equality of the originating clock valuation with κ is given by the fact that no time passes during the transition, and therefore both valuations are equal except on a set of newly-reset clocks, which value is always 0 and does not modify zone-satisfiability.

We now prove the reciprocal (or completeness) part of the theorem. It consists in showing that, starting from concrete bisimilarity, we obtain a family of abstract bisimulations indexed on name constraints. This abstract bisimulation is defined in two steps:

$$\begin{aligned} S_{nc} \triangleq \{(t\,\zeta, u\,\xi)^{\mu} \mid\ & \zeta \in Reg(\mathrm{fn}(t)),\ \xi \in Reg(\mathrm{fn}(u)), \mu \in MCE_{\mathrm{fn}(t)\cup\mathrm{fn}(u)}(\top) \\ & (\forall \rho, \kappa, \alpha.\ (\rho \vDash (\zeta^{\downarrow x\uparrow} \wedge \zeta) \wedge \kappa \vDash (\xi^{\downarrow x\uparrow} \wedge \xi) \wedge \rho(x) = \kappa(x) \wedge \alpha \vDash \mu \wedge \\ & (\zeta^{\downarrow x\uparrow} \wedge \zeta)^{/x} \Leftrightarrow (\xi^{\downarrow x\uparrow} \wedge \xi)^{/x}) \Rightarrow (t\alpha)\rho^{\backslash x} \sim_{tl} (u\alpha)\kappa^{\backslash x})\} \end{aligned}$$

Is an elementary set that contains only *regions* (and *not* zones); it is nearly a bisimulation, but it lacks the capacity of associating zones that are reached through time-progression. For that reason, we shall show that the *time closure* extension of S_{nc}, noted S and defined next, *is* a bisimulation.

$$\begin{aligned} S \triangleq S_{nc} \cup \{(t\,\zeta'', u\,\xi'')^{\mu} \mid\ & \exists t, u, \zeta, \xi, \mathcal{S}.\ (t\,\zeta, u\,\xi)^{\mu} \in S_{nc} \wedge t\,\zeta^{\downarrow x} \xrightarrow{\mathcal{S}} t'\,\zeta' \wedge u\,\xi^{\downarrow x} \xrightarrow{\mathcal{S}} u'\,\xi' \wedge \\ & (\zeta'^{/x} \Leftrightarrow \xi'^{/x}) \wedge (\zeta'' \Leftrightarrow \zeta'^{\backslash x}) \wedge (\xi'' \Leftrightarrow \xi^{\backslash x})\} \end{aligned}$$

We shall note $S^{\mu} \triangleq \{(t\,\zeta, u\,\xi) \mid (t\,\zeta, u\,\xi)^{\mu} \in S\}$. We shall also take the liberty to employ either S or $S_{stl} \triangleq \bigcup_{\mu} \{\{S^{\mu}\}^{\mu}\}$ when the distinction between them is unimportant.

We now show that S (or S_{stl}) is a symbolic timed late bisimulation. The proof is, as before, by induction over the structure of terms, employing (this time in that order) the soundess and the weak completeness theorems.

Let us therefore suppose that given a name constraint $\mu \in MCE_{\mathrm{fn}(t)\cup\mathrm{fn}(u)}(\top)$ and two regions ζ and ξ such that $(\zeta^{\downarrow x\uparrow} \wedge \zeta)^{/x} \Leftrightarrow (\xi^{\downarrow x\uparrow} \wedge \xi)^{/x}$, we have for all ρ, κ, α satisfying $\rho \vDash (\zeta^{\downarrow x\uparrow} \wedge \zeta) \wedge \kappa \vDash (\xi^{\downarrow x\uparrow} \wedge \xi) \wedge \rho(x) = \kappa(x) \wedge \alpha \vDash \mu$, that $((t\alpha)\rho^{\backslash x} \sim_{tl} (u\alpha)\kappa^{\backslash x})$. Obviously, $(t\,\zeta, u\,\xi) \in S^{\mu}$. We hence prove that all actions of $t\,\zeta$ are matched by $u\,\xi$ and lead to bisimilar pairs in S. We start by considering abstract time-passing transitions, forgetting about renamings for the sake of tractability.

Take $t\,\zeta \xrightarrow{S} t'\,\zeta'$. We first employ the soundness theorem: for any $\rho_s \vDash \zeta$ and $\delta > 0$ with $\rho_s^{+\delta} \vDash ((\zeta^{\nearrow} \backslash \zeta^{=\swarrow}) \wedge \zeta'^{\swarrow})$, we obtain a concrete transition $t\rho_s \xrightarrow{\delta\,S} t\rho_s^{+\delta}$. We have therefore a concrete transition for all $\rho^{\backslash x}$ such that $\rho \vDash (\zeta^{\downarrow x\nearrow} \wedge \zeta)$, because $(\zeta^{\downarrow x\nearrow} \wedge \zeta)^{\backslash x} \Leftrightarrow \zeta$. Since $t\rho^{\backslash x} \sim_{tl} u\kappa^{\backslash x}$ and $(\zeta^{\downarrow x\nearrow} \wedge \zeta)^{/x} \Leftrightarrow (\xi^{\downarrow x\nearrow} \wedge \xi)^{/x}$, $u\kappa^{\backslash x}$ has a matching term $u\kappa^{\backslash x} \xrightarrow{\delta\,S} u'\kappa'$ for *all* $\kappa \vDash \xi$. We can now use the weak completeness theorem to show that $u\,\xi$ has a matching symbolic transition for any concrete transition. From this results the existence of a zone ξ_c such that $\kappa^{\backslash x} \vDash \xi_c$ and there is a symbolic transition for all the concrete transitions that lead outside of the zone ξ_c itself. Furthermore, we can take $\xi \Leftrightarrow \xi_c$, since $\kappa^{\backslash x} \vDash \xi$, and therefore ξ_c contains ξ and we can then show that the weak completeness theorem can be applied again on ξ. The completeness theorem however allows certain concrete transitions not to be matched by any abstract transition, in the case where the concrete transition takes place between instants that belong to the same abstract zone. There are precisely three cases:

- if ξ and ζ are regions and $\xi^{/x} \Leftrightarrow (x = q)$ for some q, then there is no $\delta > 0$ so that $u\kappa^{+\delta}$ may remain in $u\,\xi$, and both $u\,\xi$ and $t\,\zeta$ have a time-passing transition for any bound-respecting δ;

- if ξ and ζ are regions and $\xi^{/x} \Leftrightarrow (q < x < q + 1)$ for some q, then $u\,\xi$ and $t\,\zeta$ have both a self-loop symbolic transition representing all the concrete transitions that take place inside a unique region, while another abstract transition represents the concrete transitions that go outside of their region of origin (the ones such that $\rho^{+\delta} \nvDash \xi$).

- if ξ and ζ are zones, then they have no self-loops to themselves, but only transitions to other zones, which are matched by definition of S; the concrete transitions inside a zone are easily matched because $(\zeta^{\downarrow x\nearrow} \wedge \zeta)^{/x} \Leftrightarrow (\xi^{\downarrow x\nearrow} \wedge \xi)^{/x}$, ensuring that for any κ there is a corresponding ρ with $\rho(x) = \kappa(x)$ (and vice-versa) with the same potential time progression.

By the definition of timed late bisimilarity, after the time-passing transition we have still $t'\,\rho' \sim_{tl} u'\,\kappa'$ for any ρ' and κ', with $\rho' \vDash \zeta'$ and $\kappa' \vDash \xi'$. The interval trajectory property (*cf.* Section 3.8) yields that all the κ satisfying ξ are reachable through a time-passing transition of $u\kappa$ for some value δ, and similarly for $t\rho$. Furthermore, by definition of S_{nc} and S, the originating as well as the destination zones (ζ and ξ, ζ' and ξ') have the same constant width because ζ and ξ are regions, and $\zeta'^{/x} \Leftrightarrow \xi'^{/x}$ in the definition of S. Hence ζ' and ξ' satisfy the condition required for $(t'\zeta', u'\xi')$ to be in S_{nc}, and therefore $(t'\zeta', u'\xi')$ belongs to S, closing the time-addressing case.

We now address the case of discrete transitions. Take ζ_i ranging over a slice partition $\zeta_1, \cdots, \zeta_k$ of $(\zeta^{\downarrow x\nearrow} \wedge \zeta)$, and suppose we have a slice partition of $(\xi^{\downarrow x\nearrow} \wedge \xi)$ with each ξ_i fulfilling the first clause of the definition of symbolic timed bisimulation. We have to prove the existence of a $(\zeta_i^{\backslash x} \wedge \xi_i^{\backslash x})$-partition $\zeta\xi_1, \cdots, \zeta\xi_l$, ranged over by $\zeta\xi_j$, where any discrete transition of $t\,\zeta\xi_j$ is matched by $u\,\zeta\xi_j$. In the case where $t\,\zeta_i$ may not perform any discrete transition, finding such a partition is trivial; just take ζ_i itself, then there is no obligation on $u\,\xi_i$ to perform any transition either. We now can, without losing generality, suppose that all $\zeta\xi_j$ are *regions*. Indeed, proving the

existence of a partition composed only of regions implies proving the existence of a partition composed of (otherwise region-grouping) zones. We henceforth suppose that each $t\,\zeta\xi_j \xrightarrow{\pi,\eta} t'\,\zeta'_j$ with $(\mathrm{bn}(\pi) \cap (\mathrm{fn}(t) \cup \mathrm{fn}(u))) = \emptyset$ (those transitions exist by consequence of the fact that $t\,\zeta_i$ can perform a transition with the same label). Then for each $\theta \in MCE_{\mathrm{fn}(t)\cup\mathrm{fn}(u)}(\mu \wedge \eta)$ we have to prove that $u\,(\zeta\xi_j)^{/clocks(\xi)} \xrightarrow{\pi',\theta'} u'\xi'_j$ with $\theta \Rightarrow \theta'$, and $\pi' \equiv^{\theta} \pi$. The soundness theorem is applicable to the transition of $t\,(\zeta\xi_j)^{/clocks(\zeta)}$ because $ports(\alpha) \subseteq (\mathrm{fn}(t) \cup \mathrm{fn}(u))$, that yields $(\mathrm{bn}(\pi) \cap (\mathrm{fn}(t) \cup \mathrm{fn}(u)) = \emptyset) \Rightarrow (\mathrm{bn}(\pi) \cap (\mathrm{fn}(t) \cup ports(\alpha)) = \emptyset)$. By this application we obtain an outgoing transition $(t\alpha)\rho^{\backslash x} \xrightarrow{\pi\alpha} t'\rho'$ for each $\rho \vDash \zeta\xi_j^{/clocks(\zeta)}$, $\rho' \vDash \zeta'_j$, and $\alpha \vDash \eta$. Since $\zeta\xi_j \Rightarrow \zeta$ and $\zeta\xi_j \Rightarrow \xi$, $(t\,\zeta, u\,\xi) \in S$ implies by the induction hypothesis that for all $\kappa \vDash \zeta\xi_j^{/clocks(\xi)}$ such that $\kappa(x) = \rho(x)$ we have $(t\alpha)\rho^{\backslash x} \sim_{tl} (u\alpha)\kappa^{\backslash x}$. The definition of timed late bisimulation then yields the existence of a transition $(u\alpha)\kappa^{\backslash x} \xrightarrow{\pi\alpha} u'\kappa'$, because $(\mathrm{bn}(\pi) \cap (\mathrm{fn}(t) \cup \mathrm{fn}(u))) = \emptyset$ implies that for any renaming α, $\mathrm{bn}(\pi) = \mathrm{bn}(\pi\alpha)$, and thus $(\mathrm{bn}(\pi\alpha) \cap (\mathrm{fn}(t) \cup \mathrm{fn}(u))) = \emptyset$.

We apply the weak completeness theorem on that transition. As consequence of the applicability of the soundness theorem we have $(\mathrm{bn}(\pi\alpha) \cap \mathrm{fn}(t\alpha)) = \emptyset$; we therefore are guaranteed by completeness of the existence of an abstract transition $u\,\xi_c \xrightarrow{\pi',\theta'} u'\,\xi'_c$, for any $\theta' \exists\, \alpha$, and ξ_c and ξ'_c satisfied by $\kappa^{\backslash x}$ and κ', respectively. We can assume that ξ_c is a region; indeed, if it is not, it suffices to take the region $\xi_c{}''$ containing $\kappa^{\backslash x}$, that satisfies $\xi_c{}'' \Rightarrow \xi_c$. The proof of the existence of $\xi_c{}''$ is then obtained in the same was that the one of ξ_c. Furthermore, because $\zeta\xi_j$ is a region, then mandatorily ξ'_j is a region too (the only potential difference between $\zeta\xi_j$ and ξ'_j is that some clocks may have been reset in ξ'_j). Hence, we obtain $\xi_c \Leftrightarrow \zeta\xi_j^{/clocks(\xi)}$ and $\xi'_c \Leftrightarrow \xi'_j$, all of them being regions, each having one instant in common with its peer.

We are now left with proving that for any $\theta \in MCE_{\mathrm{fn}(t)\cup\mathrm{fn}(u)}(\mu \wedge \eta)$ we have $\theta \Rightarrow \theta'$ (yielding by the way $\pi \equiv^{\theta} \pi'$). Actually, as for any $\alpha \vDash \eta$ we can take any value for θ' such that $\alpha \vDash \theta'$ to obtain an symbolic transition, there is one conspicuous value that we can take for all those θ', that is η. We have then found a matching transition of $u\,\zeta\xi_j$, because any $\theta \in MCE_{\mathrm{fn}(t)\cup\mathrm{fn}(u)}(\mu \wedge \eta)$ trivially implies η. Now, by the definition of timed late bisimilarity, we have $t'\rho' \sim_{tl} u'\kappa'$ for all such reachable processes; and because no clock value is modified other that the ones that have been potentially set to 0, we have indeed $\rho'(x) = \kappa'(x)$. As furthermore θ'' (as given in Definition 4.2.3) implies η, then any $\gamma \vDash \theta''$ also fulfills $\gamma \vDash \eta$, and any t' (resp. u') is such that there exists t'' (resp. u'') with $t' \equiv t''\gamma$ (resp. $u' \equiv u''\gamma$). Hence $(t\,\zeta'_i, u\,\xi'_i) \in S^{\theta''}$; this concludes the proof.

□

Proof of Theorem 4.2.4 (Soundness of the Proof System).

We actually prove that the proof system decides bisimulation up-to a constraint ϕ, noted $\mu : \phi, \top \rhd t\,\zeta = u\,\xi$. The proof for bisimulation up-to infinity is obtained by taking $\phi \Leftrightarrow \top$. The proof goes as usual at three levels: at a most general level by induction on the derivation trees and reasoning on the proof rule that is used last, at a secondary level by co-induction on the transitions triggerable by processes (to prove bisimilarity of processes associated by the proof system), and at a third level

by performing structural induction on terms and reasoning on the semantic rule that is used last to infer the existence of a transition.

Let us first consider the rule P-Input. Its antecedent implies that $\mu \rhd t\,\zeta = u\,\xi$, yielding $t\,\zeta \sim^{\mu}_{stl} u\,\xi$ by the induction hypothesis. The rule allows to infer that

$$\mu : \phi, \psi_{\text{TMISS}} \rhd ([\sigma, \upsilon]a(c).\,t)\zeta = ([\sigma', \upsilon']b(c).\,u)\xi\ .$$

By the antecedent $Equ_{Tpref}(\mu, t_{a(c)}, u_{b(c)}, L)$, both processes are equivalence classes, each allowing either none or a unique discrete transition, this transition being the reduction of the prefixes given by the rule PRE of Table 4.2. In such a case we have therefore $([\sigma, \upsilon]a(c).\,t)\zeta \xrightarrow{a(c),\top} t\,\zeta$ and $([\sigma', \upsilon']b(c).\,u)\xi \xrightarrow{b(c),\top} u\,\xi$. By the definition of symbolic timed late bisimulation, those two terms are discrete-bisimilar in environment μ, because $\mu \Rightarrow (a = b)$ by the side condition of rule P-Input.

Now, to deal with time-passing transitions, we take a deeper look into the predicate Equ_{Tpref}. Indeed, besides allowing the same discrete transitions for the two terms, it also enforces their time-wise up-to-ϕ bisimilarity. In case a time progression may be allowed by ϕ (that is $((\zeta^{\uparrow} \wedge \phi_{\zeta}) \not\Leftrightarrow \zeta) \ \wedge\ ((\xi^{\uparrow} \wedge \phi_{\xi}) \not\Leftrightarrow \xi)$), it forces ζ and ξ to have the same constant width and to allow exactly the same time-passing transitions by allowing t and u to invoke only the same axiom. All of this implies that $t\,\zeta_i$ and $u\,\xi_i$ are equivalence classes and satisfy the first clause of Definition 4.2.3: for any slice partition $\zeta_1, \cdots, \zeta_k$ of ζ there must be a slice partition $\xi_1, \cdots, \xi_k$ of ξ such that each ζ_i corresponds to ξ_i in that they have the same constant width, their time-passing transitions are identical modulo bisimilarity, and the resulting zones are bisimilar. Hence, all ζ_i and ξ_i have the same discrete transitions; all the ζ_i and ξ_i that are reachable from another ζ'_i and ξ'_i (using the TREG or TREQ rule) are therefore bisimilar. For time-passing transitions that reach a zone not part of ζ or ξ, there are two possibilities:

- the reached zone can be the higher border of a time continuity zone, as given by the semantic rules of Tables 4.3 and 4.4, or
- this zone can be the zone comprised between the lower border of the current zone and the upper bound given by ϕ.

The last clause in Equ_{Tpref} ensures that, in any case, those zones must be equivalent according to the value of a fictitious new clock x, and that the resulting processes must be bisimilar up-to-ϕ. This completes the proof of the P-Input case. We make two remarks. The first one is that another reasonable solution avoiding this complexity could have been adopted, that would be to allow only *regions* for ζ and ξ, and to force the use of rule Part-ζ for zones. This latter reasoning however introduces another complexity, due to that use of Part-ζ; we preferred to use the first solution, at the cost of one additional definition. The second remark is that the constraint ψ takes the value ψ_{TMISS} in the proved judgment. It is when ψ_{TMISS} becomes false, and only at that moment, that the time is prevented from progressing by deadlock of either $([\sigma, \upsilon]a(c).\,t)\zeta$ or $([\sigma', \upsilon']b(c).\,u)\xi$.

The cases for the following rules: C-Input, Output, Tau and Error, are very similar to the P-Input case, and we shall not treat them in detail. We instead treat the rule Reset. This case is very simple, since it forces to check the bisimilarity of

$t\,\zeta^{\downarrow x}$ and $u\,\xi^{\downarrow x}$ to infer the bisimilarity of $(\nu x\ t)\zeta$ and $(\nu x\ u)\xi$. The only applicable rule, for both discrete and time-passing transitions, is the rule RESET, that precisely states that the transitions of a process $(\nu x\ t)\zeta$ are precisely the ones of $t\,\zeta^{\downarrow x}$. The condition of applicability of Reset, that $x \notin (clocks(\zeta) \cup clocks(\xi))$, ensures that RESET is usable for both t and u.

Proving the soundness of the Choice rule is the most interesting case (and the most complex, relatively). We have by the induction and coinduction hypotheses the existence of two bisimulations, $t\,\zeta \sim_{stl} u\,\xi$, and $t'\,\zeta \sim_{stl} u'\,\xi$. Now suppose the Choice rule applies; we have to prove that $(t+t')\zeta \sim_{stl} (u+u')\xi$. The hypothesis $t\,\zeta \sim_{stl} u\,\xi$ implies that, for any member ζ_i of a slice partition $\zeta_1, \cdots, \zeta_k$ of $(\zeta^{\downarrow x\uparrow} \wedge \zeta)$, there is a ξ_i member of a slice partition $\xi_1, \cdots, \xi_k$ of $(\xi^{\downarrow x\uparrow} \wedge \xi)$ with $const\_width(\zeta_i) = const\_width(\xi_i)$, such that there exists a $(\zeta_i \wedge \xi_i)$-partition $\zeta\xi_1, \cdots, \zeta\xi_l$ ranged over by $\zeta\xi_j$, where the transitions of $t\,\zeta\xi_j$ are matched by the ones of $u\,\zeta\xi_j$. From $t'\,\zeta \sim_{stl} u'\,\xi$, there is a similar statement for the same ζ_i that yields a slice partition $\xi'_1, \cdots, \xi'_{k'}$ and an associated $(\zeta_i \wedge \xi'_i)$-partition $\zeta\xi'_1, \cdots, \zeta\xi'_{l'}$ ranged over by $\zeta\xi'_{j'}$, such that the transitions of $t'\zeta\xi'_{j'}$ are matched by the ones of $u'\zeta\xi'_{j'}$.

We first solve the case of symbolic time-passing transitions. We do so in two steps, first disregarding the conditions ϕ and ψ of the proof rules and focusing on the transitions themsleves, and only after this examinig the up-to aspects. The semantic rule that can be used to infer a time-passing transition for $(t + t')\zeta_i$ is TSUM. It imposes that both t and t' be able to perform transitions to zones at a similar distance (in time) for a transition of the sum to happen. Suppose therefore that $t\,\zeta_i^{\downarrow y} \xrightarrow{\mathcal{S}} t''\,\zeta'$ and $t'\,\zeta_i^{\downarrow y} \xrightarrow{\mathcal{S}'} t'''\,\zeta''$ with the side-condition of TSUM respected. TSUM then applies, yielding a transition $(t+t')\zeta_i \xrightarrow{\mathcal{S},\mathcal{S}'} (t''+t''')(\zeta' \wedge \zeta'')$. Because $y \notin \{clocks(\zeta) \cup clocks(\xi)\}$, we have by congruence that $t\,\zeta_i^{\downarrow y}$ and $u\,\xi_i^{\downarrow y}$ perform the same transitions as $t\,\zeta_i$ and $u\,\xi_i$. We have then, by $t\,\zeta \sim_{stl} u\,\xi$ and $t'\,\zeta \sim_{stl} u'\,\xi$, that for any $1 \leq i \leq k$, $u\,\xi_i^{\downarrow y} \xrightarrow{\mathcal{S}''} u''\,\xi'$ and $u'\,\xi_i'^{\downarrow y} \xrightarrow{\mathcal{S}'''} u'''\,\xi''$ with $\mathcal{S} \preceq \mathcal{S}''$ and $\mathcal{S}' \preceq \mathcal{S}'''$. We have also for the same reason $const\_width(\zeta_i) = const\_width(\xi_i) = const\_width(\xi'_i)$, $t\,\zeta_i^{\setminus x} \sim_{stl} u\,\xi_i^{\setminus x}$, $t'\,\zeta_i^{\setminus x} \sim_{stl} u'\,\xi_i'^{\setminus x}$, $\zeta'^{/x} \Leftrightarrow \xi'^{/x}$, $\zeta''^{/x} \Leftrightarrow \xi''^{/x}$, $t''\,\zeta' \sim_{stl} u''\,\xi'$, and $t'''\,\zeta'' \sim_{stl} u'''\,\xi''$.

From the slice partitions $\xi_1, \cdots, \xi_k$ and $\xi'_1, \cdots, \xi'_k$ we have to find a slice partition of ξ that is able to match the time-passing transitions of $(t+t')\zeta_i$. This is easy, since by definition of timed late bisimulation, ξ_i and ξ'_i are equivalent: they have the same constant width and, being members of a slice partition of ξ, they are such that $\xi_i^{\uparrow\downarrow} \Leftrightarrow \xi^{\uparrow\downarrow}$ and $\xi_i'^{\uparrow\downarrow} \Leftrightarrow \xi^{\uparrow\downarrow}$; this fact ensures notably that the fractional parts of the clocks of ξ_i and ξ'_i are sorted in the same way. The applicability of TSUM to $(u+u')\xi_i$ (we ommit ξ'_i for the sake of simplicity) is obtained by showing that $\xi'^{/y} \Leftrightarrow \xi''^{/y}$. The bisimilarities $t''\,\zeta' \sim_{stl} u''\,\xi'$ and $t'''\,\zeta'' \sim_{stl} u'''\,\xi''$ given above guarantee the existence of a clock $z \notin \{clocks(\zeta') \cup clocks(\xi')\}$ such that $(\zeta'^{\downarrow z\uparrow} \wedge \zeta')^{/z} \Leftrightarrow (\xi'^{\downarrow z\uparrow} \wedge \xi')^{/z}$ and $(\zeta''^{\downarrow z\uparrow} \wedge \zeta'')^{/z} \Leftrightarrow (\xi''^{\downarrow z\uparrow} \wedge \xi'')^{/z}$. Similarly, from $\zeta'^{/x} \Leftrightarrow \xi'^{/x}$ we obtain $(\zeta'^{\downarrow z\uparrow} \wedge \zeta')^{/x} \Leftrightarrow (\xi'^{\downarrow z\uparrow} \wedge \xi')^{/x}$. Then, by $\zeta_i \Rightarrow (\zeta^{\downarrow x\uparrow} \wedge \zeta)$ and $\zeta' \Rightarrow (\zeta_i^{\downarrow y})^{\uparrow}$ (meaning that ζ' is obtained from ζ_i after some time progression), we get the existence of some p and q, such that $x - y > p$ and $y - z > q$ (the fractional part of y is comprised between the one of x and the one of z). This leads to $(\zeta'^{\downarrow z\uparrow} \wedge \zeta')^{/y} \Leftrightarrow (\xi'^{\downarrow z\uparrow} \wedge \xi')^{/y}$. A similar reasoning yields also to $(\zeta''^{\downarrow z\uparrow} \wedge \zeta'')^{/y} \Leftrightarrow (\xi''^{\downarrow z\uparrow} \wedge \xi'')^{/y}$. The side-condition of the

TSUM rule applied on $t\,\zeta_i$ and $t'\,\zeta'_i$ implies that $\zeta'^{/y} \Leftrightarrow \zeta''^{/y}$; we get trivially after the reset of a clock z that $(\zeta'^{\downarrow z\uparrow} \wedge \zeta')^{/y} \Leftrightarrow (\zeta''^{\downarrow z\uparrow} \wedge \zeta'')^{/y}$. We thus have by implication $(\xi'^{\downarrow z\uparrow} \wedge \xi')^{/y} \Leftrightarrow (\xi''^{\downarrow z\uparrow} \wedge \xi'')^{/y}$. Now by taking z out, we get the original zones: $(\xi'^{\downarrow z\uparrow} \wedge \xi')^{\backslash z} \Leftrightarrow \xi'$ and $(\xi''^{\downarrow z\uparrow} \wedge \xi'')^{\backslash z} \Leftrightarrow \xi''$. We hence finally have $\xi'^{/y} \Leftrightarrow \xi''^{/y}$ and we may apply TSUM on $u\,\xi_i^{\downarrow y}$ and $u'\,\xi_i^{\downarrow y}$. We obtain $(u+u')\xi_i \xrightarrow{\mathcal{S}'',\mathcal{S}'''} (u''+u''')(\xi' \wedge \xi'')$. The first clause of Definition 4.2.3 is then respected because the union of ready-sets is monotonous regarding $\preceq$, yielding $(\mathcal{S},\mathcal{S}') \preceq (\mathcal{S}'',\mathcal{S}''')$.

We now have to show that $(u''+u''')(\xi' \wedge \xi'') \sim_{stl} (t''+t''')(\zeta' \wedge \zeta'')$; this is done coinductively, by showing that we can derive $\mu \rhd (u''+u''')(\xi' \wedge \xi'') = (t''+t''')(\zeta' \wedge \zeta'')$. We shall not give all the details, but the inference tree for equaring those terms is nearly the same as in the case we just proved. Basically, only a clock reset may have been supressed from t and t' to obtain t'' and t''' (and similarly for u'' and u'''). Thus, any application of the Reset rule may be suppressed, while all the other rules apply in the same way, this being provable by induction on the structure of terms. A funtamental role is hold again by the last clause of antecedent Euq_R in the Input axiom, guaranteeing the bisimilarity of terms after any time-passing transition.

Now, at the light of this latter assertion, we take the up-to conditions into account. We have to show

$$\mu : (\phi \wedge \phi'), (\psi \wedge \psi') \rhd (t+t')\zeta = (u+u')\xi$$

from $\mu : (\phi \wedge \psi'), \psi \rhd t\,\zeta = u\,\xi$ and $\mu : (\phi' \wedge \psi), \psi' \rhd t'\,\zeta = u'\,\xi$. We have first to justify the fact that proving the equality of the component processes above up-to $(\phi \wedge \psi')$ and $(\phi' \wedge \psi)$, respectively, is sufficient to prove the equality of the composed processes up-to $(\phi \wedge \phi')$ with deadlocking limit $(\psi \wedge \psi')$. This deadlocking limit is logically obtained by conjunction of the individual deadlocking limits of $t\,\zeta$ and $t'\,\zeta$: as soon as $t\,\zeta$ has no time-passing transition, then $(t+t')\zeta$ may not perform any time-passing transition either, and similarly for $t'\,\zeta$. Now, we can use the fact that one branch of a choice will deadlock to avoid performing a complete examination of the other. Indeed, to prove the equality of the composed processes up-to $(\phi \wedge \phi')$, it is only necessary to check $\mu : (\phi \wedge \psi'), \psi \rhd t\,\zeta = u\,\xi$ (and similarly for t', u'). By the antecedent Equ_R of the rule producing this latter judgement, $t\,\zeta$ and $u\,\xi$ have the same time-progression capabilities (including deadlock), up-to the time where ϕ becomes false or t', u' block time progression. More precisely, either $\psi' \Rightarrow (\phi \wedge \psi)$ and both t' and u' will block time progression in $(t+t')\zeta$ and $(u+u')\xi$, making irrelevant the need to examine further time progression for t and u, or $\psi' \Rightarrow (\phi \wedge \psi)$ and time progression must be considered for t and u either until the specified limit ϕ or until time blocks, invalidating ψ.

In any case, to complete the induction step, we have now to prove that

$$(t''+t''')(\zeta' \wedge \zeta'') \sim_{stl} (u''+u''')(\xi' \wedge \xi'')$$

up-to $(\phi \wedge \phi')$ with deadlock at $(\psi \wedge \psi')$. This is done by proving that we can derive

$$\mu : (\phi \wedge \phi'), (\psi \wedge \psi') \rhd (t''+t''')(\zeta' \wedge \zeta'') = (u''+u''')(\xi' \wedge \xi'')$$

in the proof system. We can indeed conditionally derive $\mu : (\phi \wedge \psi'), \psi \rhd t''\,\zeta' = u''\,\xi'$ and $\mu : (\phi' \wedge \psi), \psi' \rhd t'''\,\zeta'' = u'''\,\xi''$ by using nearly the same derivation tree as before.

The situation depends on the previously accomplished time-passing transitions of $(t + t')\zeta$; there are four cases to consider:

1. the upper border of $(\phi \wedge \phi')$ has been reached,
2. the upper border of $(\psi \wedge \psi')$ has been reached,
3. the time-passing transition of $(t+t')\,\zeta$ has crossed a time-continuity boundary by reaching the upper limit of the current equivalence class (without invalidating $(\phi \wedge \phi')$ or $(\psi \wedge \psi')$),
4. the transition has let the processes remain in the same equivalence class.

In the first case, to prove time-bisimilarity up-to ϕ is equivalent to proving discrete-bisimilarity for the current zone, since time-passing transitions of $(t+t')\zeta$ that may invalidate $(\phi \wedge \phi')$ are not required to be matched by $(u + u')\xi$. In the second case, whenever one component process reaches its time-progression limit, then the composed process is time-blocked too. There is therefore no more time-passing transition for neither $(t + t')\zeta$ nor $(u + u')\xi$, which are hence time-wise bisimilar. The third and fourth cases are dealt with in a similar fashion. In those cases, we can infer the equality of component processes using the same derivation tree as before, except that the last step may be taken out: processes t'' and t''' may differ from t and t' because a clock-reset may have been exercised during the accomplished time-passing transition (and similarly for u'' and u'''). In the fourth case, the inferred transitions have the same ready-sets as before, which differ in the third case. In the third case, the last clause of condition Equ_R used in the antecedent of the axioms guarantees that the reached limit of time continuity only allows bisimilar time-passing transitions. This closes the time-bisimilarity analysis.

The second clause of the bisimilarity definition, that imposes discrete transitions to match each other, is solved by adapting the classical uni-directional reasoning used on CCS and the π-calculus (the reasoning in the opposite direction can be inferred straightforwardly from the presented one). We show that, the partitioning $\zeta_1, \cdots, \zeta_k$ of ζ being fixed, there is at least one partitioning of ξ such that both clauses of Definition 4.2.3 are fulfilled. As shown in the timing-based analysis, the two potential matching partitions of ξ that can be used are actually identical, each ξ_i being equivalent to ξ'_i. We can therefore use either $\zeta\xi_1, \cdots, \zeta\xi_l$ or $\zeta\xi'_1, \cdots, \zeta\xi'_{l'}$; indifferently, we shall choose the first one. We shall also assume (without loss of generality) that all $\zeta\xi_j$ and $\zeta\xi'_{j'}$ are regions: proving the existence of a "good" partition made of zones implies indeed proving the existence of a similar partition made only of regions. A region-wise partition is the finest possible way of partitioning the state space, allowing each region to have a different set of transitions. Suppose now that $(t + t')\zeta\xi_j \xrightarrow{\pi,\eta} t''\,\zeta'$ for some $\zeta\xi_j$ belonging to $\zeta\xi_1, \cdots, \zeta\xi_l$. Whenever such a discrete transition happens, it may only be inferred through rule SUM in Table 4.2. It means that either $t\,\zeta\xi_j \xrightarrow{\pi,\eta} t''\,\zeta'$, or $t'\,\zeta\xi_j \xrightarrow{\pi,\eta} t''\,\zeta'$. If $t\,\zeta\xi_j$ performs the transition, then by $t\,\zeta \sim_{stl} u\,\xi$ we trivially have a bisimilarity-compliant transition $u\,\zeta\xi_j \xrightarrow{\pi',\eta'} u''\,\xi'$ with $t''\,\zeta' \sim_{stl} u''\,\xi'$. Now if $t'\,\zeta\xi_j$ performs the transition, we obtain that $u'\,\zeta\xi_j$ is able to perform a bisimilarity-compliant transition, because $\zeta\xi_j$ is a region and thus also belongs to $\zeta\xi'_1, \cdots, \zeta\xi'_{l'}$, and $t'\zeta \sim_{stl} u'\,\xi$ yields the wanted transition. In both cases,

the resulting term contains only one of the components (the other one is eliminated by choice); the bisimilarity of either component thus gives the bisimilarity of the whole. This closes the discrete case.

The rule Conseq-ϕ allows strenghtening the up-to condition, while relaxing the dealocking limit. Its soundness can be easily seen: if we have shown that $t\,\zeta = u\,\xi$ up-to ϕ, then those processes are equivalent for all the clock valuations satisfying ϕ' when $\phi' \Rightarrow \phi$. Conversely, if we have inferred that $t\,\zeta$ and $u\,\xi$ will deadlock for all valuations such that ψ is false, then those processes will deadlock also for all valuations such that ψ' is false, if $\psi \Rightarrow \psi'$.

Finally, we treat the Part-ζ case. The proof still goes by induction. Basically, we have to show that from two slice partitions $\xi_1, \cdots, \xi_k$ and $\xi'_1, \cdots, \xi'_k$, of respectively ξ and ξ', we can build a slice partition of $(\xi \vee \xi')$ fulfilling the requirements of Definition 4.2.3. This is easy because of the side condition on the rule: the two zones must have an empty intersection, and be contiguous. The latter condition ensures that $(\xi \vee \xi')$ is indeed a zone constraints (all zone constraints have to be a conjunction of contiguous zones). The first condition implies then that $(\xi \vee \xi')$ has a constant width: if neither ξ nor ξ' intersects with the time-progression area of the other, then whenever a diagonal line of the form

$$((\bigwedge_{y,y' \in clocks(\zeta)} (y - y' = q_{y,y'})) \wedge \zeta)$$

intersects $(\xi \vee \xi')$, then the intersection of this diagonal with $(\xi \vee \xi')$ is either fully included in ξ or fully included ξ'. From Definition 4.2.2 and the last clause of side condition of rule Part-ζ imposing that $const\_width(\xi) = const\_width(\xi')$, the width of $\xi \vee \xi'$ is thus constant. For any slice partition of ζ ranged over by ζ_i, we consider the slice partition of $\xi \vee \xi'$ defined for each i by $\xi_i \vee \xi'_i$, with ξ_i and ξ'_i associated to ζ_i in the respective partitions of ξ and ξ' corresponding to that partition of ζ. The members of this partition then respect the requirements imposed on both time-passing and discrete transitions by Definition 4.2.3. The first clause of the definition is trivially respected, the time-passing transitionsbeing by assumption identical for $t\,\zeta_i$, $t\,\xi_i$, and $t\,\xi'_i$. For the discrete transitions, the existence of partitions $\zeta\xi_1, \cdots \zeta\xi_l$ of $(\zeta_i \wedge \xi_i)$ and $\zeta\xi'_1, \cdots \zeta\xi'_{l'}$ of $(\zeta_i \wedge \xi'_i)$ imply the existence of such a partition for $(\zeta_i \wedge (\xi_i \vee \xi'_i))$. We can as before suppose that all $\zeta\xi_j$ and $\zeta\xi'_j$ belonging to those partitions are regions, since the existence of a zone-comprising partition implies the existence of a partition made only of regions, obtained by further dividing existing zones until none remain. The intersection between ξ_i and ξ'_i being empty, we can take the union of those partitions $\zeta\xi_1, \cdots \zeta\xi_l, \zeta\xi'_1, \cdots \zeta\xi'_{l'}$ to form a region partition of $(\zeta_i \wedge (\xi_i \vee \xi'_i))$. This closes the case for discrete partitions.

We shall not treat the Absurd-ζ axiom, which is trivial. We shall not either address in details the rules of Table 4.2, that deal with naming issues. The naming issues have almost completely been overlooked in the previous reaonings, the rules of Table 4.6 making no use of the naming context μ (except for Input, which has received the adapted treatment). The proofs of rules Res, Match, and Part-μ can be found in the already-cited papers by Lin [Lin94]. □

Proof of Theorem 4.2.5 (Completeness of the Proof System). Suppose $(t\,\zeta, u\,\xi) \in R^\mu$ for some symbolic bisimulation R^μ. The proof goes as usual by putting the processes

in standard form using the *bound output* derived action prefix. Our processes however do not fully comply with this method, since the clock reset operators may not be moved at will (as it is the case for port name restriction), and *error* terms can occur. The standard form for (error-free) π^δ processes terms t and u is thus a bit more complex:

$$t = \nu x \sum_i [\mu_i^t][\sigma_i^t, \upsilon_i^t]\pi_i^t.t_i \text{ and } u = \nu x \sum_j [\mu_j^u][\sigma_j^u, \upsilon_j^u]\pi_j^u.u_j$$

where x is a clock names such that $(\{x\} \cap (\mathrm{fc}(t) \cup \mathrm{fc}(u))) = \emptyset$. We shall treat processes with error terms separately, for tractability purpose; the treatment that shall be applied to them is however very similar (at least in essence) to the one provided to error-free terms. Roughly, to obtain the format above for erro-free terms, we can use the following lemma, easy to prove by using the proof system, performing an induction on the structure of terms:

$$\begin{array}{ll} \vdash \mu \rhd ((\nu x\ t) + u)\,\zeta = (\nu x\ (t+u))\,\zeta & \text{if } x \notin clocks(u) \\ \vdash \mu \rhd u\,\zeta = (\nu x\ u)\,\zeta & \text{if } x \notin clocks(u) \end{array}$$

Then we can perform an induction, with the height of terms as (ever-decreasing) measure, the height of a term being the greatest number of prefixes this term may sequentially reduce before terminating (all processes terminate, since we forbid the use of recursion). The base case is trivial. For the induction case, we shall use the adapted induction hypothesis when needed. We have to derive in our proof system the equality of t and u written as above, which amounts to showing

$$\mu' \rhd (\sum_i [\mu_i^t][\sigma_i^t, \upsilon_i^t]\pi_i^t.t_i)\zeta^{\downarrow x} = (\sum_j [\mu_j^u][\sigma_j^u, \upsilon_j^u]\pi_j^u.u_j)\xi^{\downarrow x}$$

after application of rule Reset, for all $\mu' \in MCE_{(\mathrm{fc}(t)\cup\mathrm{fc}(u))}(\mu)$. The matching constraint μ' belongs to the set of maximally consistent extensions of μ, that are such that $\mu' \Rightarrow \mu$ and whatever $y, z \in (\mathrm{fc}(t) \cup \mathrm{fc}(u))$ then either $\mu' \Rightarrow (y = z)$ or $\mu' \Rightarrow (y \neq z)$.

We have to show that the proof system is always able to accommodate for all bisimilar processes. The issues related to port names are solved in [Lin94]. We largely forget them and focus on the timing-related issues. In the following case analysis, we shall assume that the partitions $\zeta_1, \cdots, \zeta_k$ (ranged over by ζ_l) and $\xi_1, \cdots, \xi_k$ (ranged over by ξ_l), resulting of $t\,\zeta \sim_{stl} u\,\xi$, respect the requirements of Definition 4.2.3 and are such that each partition $\zeta\xi_1, \cdots, \zeta\xi_l$ of $\zeta_l \wedge \xi_l$ fulfilling the second clause of Definition 4.2.3 is of constant width. The existence of such partitions is guaranteed by taking each ζ_l to be of width inferior to 1, that is to have no $\zeta_l' \Rightarrow \zeta_l$ with $\zeta_l'^{\uparrow\downarrow} \Leftrightarrow \zeta^{\uparrow\downarrow}$, and $const\_width(\zeta_l') < const\_width(\zeta_l)$. Then, by $const\_width(\xi_l) = const\_width(\zeta_l)$, we get that the partitions of $(\zeta_l \wedge \xi_l)$ can only be of constant with, inferior to 1. The proofs of equivalence then depends on the transitions that may be accomplished next, by the $\zeta\xi_1, \cdots, \zeta\xi_k$ for the discrete transitions, and by the $\zeta_1, \cdots, \zeta_k$ and $\xi_1, \cdots, \xi_k$ for time-passing transitions.

We examine the case of $t\,\zeta_l$ and $u\,\xi_l$ for a given l; $\zeta\xi_1, \cdots, \zeta\xi_k$ then partitions $\zeta_l \wedge \xi_l$, and it is ranged over by $\zeta\xi_h$. the possible transitions to consider are $t\,\zeta_l \xrightarrow{S} t'\,\zeta'$ (meanwhile $u\,\xi_l \xrightarrow{S'} u'\xi'$), and $t\,\zeta\xi_h \xrightarrow{\pi_u, \mu_t} t'\,\zeta'$ (meanwhile $u\,\zeta\xi_h \xrightarrow{\pi_u, \mu_u} u'\xi'$);

indeed in each case both processes have to perform the transition when one does, by Definition 4.2.3. For each such $t\,\zeta\,\xi_h$ and $u\,\zeta\xi_h$, we can build a proof tree by structural induction on terms, separating branches of the choice with the Choice rule, and then invoking the axioms P-Input, C-Input, Output, and Tau (also Error if *Error* terms are allowed). For any branch t_i of the choice, there is therefore a corresponding branch that matches its discrete transition, and that we will also call u_i for simplicity. The axiom that can be used in such a case depends on the label of the transition.

If we take the case of port input, then the label is $\pi_t = a(c)$. We show that $\mu' \rhd t_i\zeta\xi_h = u_i\zeta\xi_h$ by application of P-Input, because the antecedent of the rule is verified. First, we have by definition of symbolic timed late bisimulation combined with the induction hypothesis that $\mu' \rhd t'\,\zeta' = u'\,\xi'$ (the height of t' and u' is inferior to the height of the unprimed terms). We then have to show that the predicate Equ_{Tpref} is verified for the given processes. This is true because:

- both $t_i\zeta\xi_h$ and $u_i\zeta\xi_h$ perform a transition with the same ready-set $\mathcal{S}$, meaning that $\zeta\xi_h \Rightarrow \sigma$ is equivalent to $\zeta\xi_h \Rightarrow \sigma'$, and similarly with υ and υ';
- $\phi \Leftrightarrow \top$ and $\psi \Leftrightarrow \bot$;
- since ζ_l and ξ_l are of constant width inferior to 1, so is $\zeta\xi_h$;
- there is at least one antecedent of a semantic rule in Table 4.3 that is true, since there is an enabled transition;
- whenever such antecedent predicate is satisfied, then by the definition of symbolic timed late bisimulation its destination zones are at the same distance in time, and again we can deduce the equivalence of the reached zones by the induction hypothesis.

The cases for the other prefix actions are very similar to this one. A proof strategy has now to be applied in order to identify all bisimilar processes that feature a choice between several branches (this strategy is however decidable). The issue is to identify braches that are not bisimilar up-to infinity, but up-to a certain limit ϕ; this is possible if and only if some other branch of the choice allows the equality to be proved with $\psi \Rightarrow \xi$ (this branch blocks the time before the branch which equality remains to be proved has a different behavior in t and u). One therefore needs to group first the branches that block time progression the most early. This being done, the solution appears easily.

Once determined the provable equality of $t\,\zeta\xi_h$ and $u\,\zeta\xi_h$ for all h, the rule Part-ζ can be applied incrementally to prove the eqality of $t\,\zeta_l$ and $u\,\xi_l$ (notice that, although the definition of timed late bisimilarity already implies the bisimilarity of those terms, it does not imply that they are provably equal, because the induction hypothesis can only be applied when the height of the terms strictly decreases). Then, the regions should be grouped on a contiguity basis by using Part-ζ, and showing that this strategy can always be respected and builds the wanted proof tree is trivial. Once the $t\,\zeta_l$ and $u\,\xi_l$ are identified, one should use again the rule part-ζ to group all the slices; the same strategy as before applies, which has now a simple

translation, that implies grouping the zone numbered from 1 to k in increasing order. This yields the completeness of the proof system.

□

Proof of Theorem 5.5.1 (Subject reduction). The statement of the subject reduction property differs in our case (as in all behavioral type systems) from Chruch's original statement in that we only have to prove the *existence* of a typing environment Γ' in which the continuation of the reduced process is well-typed, not the conservation of the type Γ under reduction. We actually prove that Γ' is a derivative of Γ by the type transition relation.

The proof of the subject reduction property however goes as usual, by structural induction on terms, with an inference on the last typing rule and the last semantic rule used to deduce the symbolic transition of the process. There is a one-to-one relation between the rules of both proof systems, since the rule to be applied last is uniquely determined in each system by the structure of the term. We address the typing rules in their order of appearance.

Suppose that the term to be typed is a bound output $([\sigma, \upsilon]\overline{a}(b^{U_1,U_2}).t)\zeta$. A discrete transition of such process may be obtained only by the axiom PRE of Table 4.2. The rule Boutput has then to be applied to infer well-typedness. This rule imposes a structure to the typing environment, that is to contain a type for the free name a. The process must conform to its type, which implies that the action prefix of the type of a matches the output on a. The rule also enforces the truthfullness of the *Next* predicate, which in this case imposes that whenever the process triggers a discrete transition, then the continuation of this process is typable in an environment that takes into account the creation of the new port name. This closes the case for discrete transitions. A time-passing transition may be obtained through one of the rules of Tables 4.3 and 4.4. The predicate *Next* then imposes the type to be capable of a comparable transition, and the resulting environment to type the continuation of the process.

The cases for free output and input go along the same line as the previous one; together, they give the base case for the structural induction. We can now use an induction hypothesis in the other cases. Showing the property for the *NewPort* rule is simple because the transitions of a process with a restriction on a port a are the same as without this restriction, but the cases of input and output transitions on subject a; the corresponding discrete transitions are no more possible when a is restricted, and the name a is taken out of the ready-sets of time-passing transitions. In both cases the proof obtained by the induction hypothesis still applies, since in the first case we have less process transitions, while the second case only concerns naming issues which are ingnored by the type system. For the *NewClock* rule the reasoning is trivial, since the transitions of $t\,\zeta^{\downarrow x}$ and $(\nu x\ t)\zeta$ are the same (by rule RESET.

Similarly, the rules *Named* and *Id* are structural rules straightforwardly solved by induction, for which the transitions of the to-be-typed process are the same as the typed process appearing in the antecedent of the rule. The rules *Sum* and *Par* associate two typing environments. In both cases the proof is immediate because:

- the time-passing transitions of the composed processes are synchronized on a

fictitious clock, ensuring similar progress capabilities for both the terms and the types,

- the discrete transitions are the ones of either t or u (in the Sum case),
- the discrete transitions are the ones of t and the ones of u (in the parallel case), plus silent transitions which are not taken into account by the typing system.

The rule for parallel composition actually hides an *assume/guarantee* reasoning, in the form of, *e.g.* , [MC81] and [AL95]. That is, whenever $\Gamma_1 \oplus \Gamma_2$ is defined, the private ports appearing in both typing environments have been checked for compatibility. It has thus been checked that the assumptions (obligations) made at any step by the type T_1 of a name a in Γ_1 will be satisfied in the future by the type T_2 of a in Γ_2. This is done in the proof system defining $Comp$ by letting T_2 let time pass until the deadline of T_1 is reached. If there is such a possibility and T_2's offers do not match, then the types are declared incompatible. The symmetric verification for T_2's obligations and T_1's offers is done at the same time. We remark again that, since our types are finite-state, the definitional axiomatization for $Comp$ is not essential to check the property, since this could be done using the usual model-checking techniques.

Now, the proof for subtyping is trivial, because the same process appears in the antecedent and in the consequent of the rule. The proof of the last rule for partitioning requires exhibiting a typing context Γ' typing the continuation of $t(\zeta \vee \zeta')$. In this case clearly, any of the two contexts typing the continuations of $t\,\zeta$ and $t\,\zeta'$ will fulfill this need. □

Proof of Theorem 5.5.2 (Type reduction). The proof of this theorem is almost symmetric to the previous one. The main motivation for the presence of this theroem is the fact that the transition of types are not in exact match with the time-passing transitions of the processes they type. In that sense, they are not "ideal" abstraction of those processes. The type reduction property however shows that the correspondence between them is tight, since both types and processes may perform reductions while preserving well-typedness. We do not repeat the argument developped when proving subject reduction. The most interesting cases are the ones of the axioms, where predicate $Next$ and $Conform$ enforce the respect of the symmetry between process reduction and type reduction, and the case for parallel composition. In the parallel case, the correctness comes from the fact that the set of transitions of well-assembled typing environments is included in the union of the transitions sets of the environments taken separately: the names of ports that have found a matching peer are typed as "non-composable" in the resulting typing environment. The validity of the rule comes therein. □

Proof of Theorem 5.5.3 (Run-time safety). The run-time safety proof consists in showing that any well-typed process may not immediately perform a (sole) time-passing transition leading to $Error$, with a precondition that the process does not contain $Error$ itself. Indeed, using the subject and type reduction proofs yields immediately the preservation of the property under discrete transitions. We have then to consider only time-passing transitions; the subject reduction yields trivially that the current process may not lead to error, since $Error$ processes are never well-typed. □

Bibliography

[ABL98] Luca Aceto, Augusto Burgueño, and Kim G. Larsen. Model checking via reachability testing for timed automata. In Bernhard Steffen, editor, *TACAS'98*, volume 1384 of *Lecture Notes in Computer Science*, pages 263–280. Springer-Verlag, 1998.

[ACD93] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, May 1993.

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–236, April 1994.

[AD96] Roberto M. Amadio and Mads Dam. Toward a modal theory of types for the π-calculus. In Bengt Jonsson, editor, *FTRTFT'96, Proceedings of 4th international symposium on Formal Techniques in Real Time and Fault Tolerant Systems, Uppsala*, volume 1135 of *LNCS*, pages 347–365. Springer, 1996.

[AdSSL+01] L. Apvrille, P. de Saqui-Sannes, C. Lohr, P. Sénac, and J.-P. Courtiat. A new UML profile for real-time system formal design and validation. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings*, volume 2185 of *LNCS*, pages 287–301. Springer, 2001.

[AF99] Luís Filipe Andrade and José Luiz Fiadeiro. Interconnecting Objects via Contracts. In Bernhard Rumpe, editor, *Proceedings UML'99 (The Second International Conference on The U nified Modeling Language)*, volume 1723 of *LNCS*, Kaiserslautern, Germany, October 1999. Springer-Verlag.

[AFD80] Krzysztof R. Apt, Nissim Francez, and Willem P. De Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(3):359–385, July 1980.

[AFH91] Rajeev Alur, Tomas Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. In *Symposium on Principles of Distributed Computing*, pages 139–152, 1991.

[AG94] R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering*, pages 71–80. IEEE Computer Society Press, May 1994.

[Agh86] G. Agha. *Actors—A Model of Concurrent Computation for Distributed Systems.* MIT Press, 1986.

[AH89] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. In *IEEE Symposium on Foundations of Computer Science*, pages 164–169, 1989.

[AH94] R. Alur and T. A. Henzinger. Real-time system = discrete system + clock variables. In T. Rus and C. Rattray, editors, *Theories and Experiences for Real-Time System Development — Papers presented at First AMAST Workshop on Real-Time System Development,* Iowa City, Iowa, November 1993, pages 1–29. World Scientific, 1994.

[AH97] Rajeev Alur and Thomas A. Henzinger. Modularity for timed and hybrid systems. In Antoni Mazurkiewicz and Józef Winkowski, editors, *CONCUR '97: Concurrency Theory, 8th International Conference*, volume 1243 of *Lecture Notes in Computer Science*, pages 74–88, Warsaw, Poland, 1–4 July 1997. Springer-Verlag.

[AH99] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design: An International Journal*, 15(1):7–48, July 1999.

[AJ95] Luca Aceto and Alan Jeffrey. A complete axiomatization of timed bisimulation for a class of timed regular behaviours. *Theoretical Computer Science*, 152(2):251–268, December 1995.

[AL93] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

[AL94] M. Abadi and L. Lamport. An old-fashioned recipe for real-time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994.

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.

[ALW89] Martín Abadi, Leslie Lamport, and Pierre Wolper. Realizable and unrealizable specifications of reactive systems. In *ICALP*, number 372 in Lecture Notes in Computer Science, Stresa, Italy, July 1989. Springer-Verlag.

[AM96] Luca Aceto and David Murphy. Timing and causality in process algebra. *Acta Informatica*, 33(4):317–350, 1996.

[AMST97] G. Agha, I. Mason, S. Smith, and C. Talcott. A foundation for actor computation. volume 7, pages 1–72, 1997.

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[AS01] Tatiana Aubonnet and Noemie Simoni. Pilote: A service creation environment in ngns. In *Intelligent Networks'2001*, Boston, USA, May 2001.

[BAL99] G. Boudol, R. Amadio, and C. Lhoussaine. The receptive distributed pi-calculus. In *FST-TCS'99*, volume 1282 of *Lecture Notes in Computer Science*, pages 304–315. Springer-Verlag, 1999.

[Bar92] H. P. Barendregt. Lambda calculi with types. In D. M. Gabbai Samson Abramski and T. S. E. Maiboum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, Oxford, 1992.

[BB89] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–73. Elsevier Science Publishers North-Holland, 1989.

[BB91] J. Baeten and J. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.

[BB92] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, Apr 1992.

[BC01] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs. In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, 2001.

[BD94] M. Boreale and R. De Nicola. A symbolic semantics for the π-calculus. *Lecture Notes in Computer Science*, 836:299–311, 1994.

[BG92] Gerard Berry and Georges Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BGS00] Sebastien Bornot, Gregor Gler, and Joseph Sifakis. On the construction of live timed systems. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 109–126, 2000.

[BH00] M. Berger and K. Honda. The two-phase commitment protocol in an extended π-calculus. In *Express'00*, volume 39 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.

[BIM95] Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can't be traced. *Journal of the ACM*, 42(1):232–268, January 1995.

[BJLY98] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi. Partial order reductions for timed systems. *Lecture Notes in Computer Science*, 1466:485–497, 1998.

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Nöel Plouzeau, and Damien Watkins. Making components contract aware. *IEEE Computer*, 32(7):38–44, July 1999.

[BK84] Jan A. Bergstra and Jan Willem Klop. The algebra of recursively defined processes and the algebra of regular processes. In Jan Paredaens, editor, *Automata, Languages and Programming, 11th Colloquium*, volume 172 of *Lecture Notes in Computer Science*, pages 82–94, Antwerp, Belgium, 16–20 July 1984. Springer-Verlag.

[BKP84] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *ACM Symposium on Theory of Computing (STOC '84)*, pages 51–63, Baltimore, USA, April 1984. ACM Press.

[BL92a] T. Bolognesi and F. Lucidi. LOTOS-like process algebras with urgent or timed interactions. In K. Parker and G. Rose, editors, *Proceedings of the 4th International Conference on Formal Description Techniques, FORTE'91*. North-Holland, 1992.

[BL92b] T. Bolognesi and F. Lucidi. Timed process algebras with urgent interactions and a unique powerful binary operator. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Proceedings REX Workshop on Real-Time: Theory in Practice,* Mook, The Netherlands, June 1991, volume 600 of *Lecture Notes in Computer Science*, pages 124–148. Springer-Verlag, 1992.

[BLS00] Beatrice Bérard, Anne Labroue, and Philippe Schnoebelen. Verifying performance equivalence for timed basic parallel processes. *Lecture Notes in Computer Science*, 1784:35–47, 2000.

[BN95] Michele Boreale and Rocco De Nicola. Testing equivalence for mobile processes. *Information and Computation*, 120(2):279–303, 1 August 1995.

[BNSL00] A. Bailly, E. Najm, J-B. Stefani, and L. Leboucher. Modélisation et vérification de protocoles temps-réel par typage comportemental. In *Proc. of CFIP'2000*, pages 183–198. Hermes, October 2000.

[Bor96] M. Boreale. Symbolic bisimulation for timed processes. *Lecture Notes in Computer Science*, 1101:321–333, 1996.

[Bou97a] G. Boudol. Typing the use of resources in a concurrent calculus. *Lecture Notes in Computer Science*, 1345:239–251, 1997.

[Bou97b] Gérard Boudol. The π-calculus in direct style. In ACM, editor, *Conference record of POPL '97, the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Paris, France, 15–17 January 1997*, pages 228–242, New York, NY, USA, 1997. ACM Press.

[BS97] G. Blair and J. B. Stefani. *Open Distributed Processing and Multimedia.* In Press. Addison-Wesley, 1997.

[BS00] Sebastien Bornot and Joseph Sifakis. An algebraic framework for urgency. *Information and Computation*, 163(1):172–202, 2000.

[BST97] Sebastien Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling urgency in timed systems. In *COMPOS*, volume 1536 of *LNCS*, pages 103–129. Springer-Verlag, 1997.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science. Springer-Verlag, 1998.

[BvW00] Back and von Wright. Contracts, games, and refinement. *Information and Computation (formerly Information and Control)*, 156, 2000.

[BW90] J. C. M. Baeten and W. P. Weijand. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, Cambridge, England, 1990.

[BZ83] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, April 1983.

[CBS+95] G. Coulson, G. S. Blair, J. B. Stefani, F. Horn, and L. Hazard. Supporting the real-time requirements of continuous media in open distributed processing. *Computer Networks and ISDN Systems*, 27(8):1231–1246, 1995.

[CCMM95] Sergio Vale Aguiar Campos, Edmund M. Clarke, Wilfredo R. Marrero, and Marius Minea. Verus: A tool for quantitative analysis of finite-state real-time systems. In *Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 70–78, 1995.

[CdO95] Jean-Pierre Courtiat and Roberto C. de Oliveira. RT-LOTOS and its application to multimedia protocol specification and validation. In *IEEE International Conference on Multimedia Networking (MmNet95)*, pages 31–45, 1995.

[Che92] L. Chen. An interleaving model for real–time systems. In Anil Nerode and Mikhail Taitslin, editors, *Proceedings of Logical Foundations of Computer Science (Tver '92)*, volume 620 of *LNCS*, pages 81–92, Berlin, Germany, July 1992. Springer.

[Cou97] Patrick Cousot. Types as abstract interpretations. In ACM, editor, *Conference record of POPL '97, the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Paris, France, 15–17 January 1997*, pages 316–331, New York, NY, USA, 1997. ACM Press.

[CPDS99] J-L. Colaço, M. Pantel, F. Dagnat, and P. Sallé. Safety analysis for non-uniform service availability in actors. In *Formal Methods for Open Object-based Distributed Systems*, February 1999.

[CPS97] J.-L. Colaço, M. Pantel, and P. Sallé. A set-constraint-based analysis of actors. In *Proceeding of the IFIP TC6 WG6.1 international workshop on Formal methods for open object-based distributed systems (FMOODS'97)*, pages 107–122. Chapman & Hall, Ltd., 1997.

[CRR02] Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. Types as models: Model checking message-passing programs. In *Conference Record of POPL'02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 45–57, Portland, Oregon, January 16–18, 2002.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):480–521, December 1985.

[dAH01a] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In Volker Gruhn, editor, *Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering (ESEC/FSE-01)*, volume 26, 5 of *SOFTWARE ENGINEERING NOTES*, pages 109–120, New York, September 10–14 2001. ACM Press.

[dAH01b] Luca de Alfaro and Thomas A. Henzinger. Interface theories for component-based design. *Lecture Notes in Computer Science*, 2211:148–160, 2001.

[dAHS02] Luca de Alfaro, Thomas A. Henzinger, and Marielle Stoelinga. Timed interfaces. In *Proceedings of the Second International Workshop on Embedded Software (EMSOFT)*, number 2491 in Lecture Notes in Computer Science, pages 108–122. Springer-Verlag, 2002.

[Dam95] Mads Dam. Compositional proof systems for model checking infinite state processes. In Insup Lee and Scott A. Smolka, editors, *CONCUR '95: Concurrency Theory, 6th International Conference*, volume 962 of *Lecture Notes in Computer Science*, pages 12–26, Philadelphia, Pennsylvania, 21–24 August 1995. Springer-Verlag.

[dB72] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[DB96] Pedro R. D'Argenio and Ed Brinksma. A calculus for timed automata. In *FTRTFT*, volume 1135 of *LNCS*, pages 110–129. Springer-Verlag, 1996.

[DFH+98] D. Donaldson, M. Faupel, R. Hayton, A. Herbert, N. Howarth, Kramer A., I. MacMillan, D. Otway, and S. Waterhouse. Dimma - a multimedia orb. In *Proc. Middleware '98*, The Low Wood Hotel, Ambleside, England, September 1998.

[DH95] D. Dill and H. Wong-Toi. Verification of real-time systems by successive over and under approximation. In P. Wolper, editor, *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939, pages 409–422, Liege, Belgium, 1995. Springer Verlag.

[DHDTS98] B. Dumant, F. Horn, F. Dang-Tran, and J.-B. Stefani. Jonathan: an open distributed processing environment in java. In *Proc. Middleware '98*, The Lake District, England, November 1998.

[DHS98] I. Demeure, F. Horn, and F. Singhoff. Automatic scheduling of a dynamic multimedia applications with polka: a case study. In *Fourth IEEE Real-Time Technology and Applications Symposium (RTAS'98)*, pages 15–19, June 1998.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[Dil89a] D. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Workshop on Automatic Verification Methods for Finite-State Systems*, June 1989.

[Dil89b] David Dill. *Trace Theory For Automatic Hierarchical Verification Of Speed-independent Circuits.* ACN Distinguished Dissertations. MIT Press, Cambridge MA, 1989.

[DK97] F. Dignum and R. Kuiper. Combining dynamic deontic logic and temporal logic for the specification of deadlines. In Jr. R. Sprague, editor, *Proceedings of thirtieth HICSS*, Wailea, Hawaii, 1997.

[DP99] Pierpaolo Degano and Corrado Priami. Non-interleaving semantics for mobile processes. *Theoretical Computer Science*, 216(1–2):237–270, 1999.

[DPCS00] F. Dagnat, M. Pantel, M. Colin, and P. Sallé. Typing concurrent objects and actors. In *L'Objet – Méthodes formelles pour les objets*, volume 6, pages 83–106, 2000.

[DTDV96] F. S. De Boer, H. Tej, W.-P. De Roever, and M. Van Hulst. Compositionality in real-time shared variable concurrency. *Lecture Notes in Computer Science*, 1135:420–433, 1996.

[EM98] F. Eliassen and S. Mehus. Type checking stream flow endpoints. In *Middleware'98, The Lake District, England*, pages 305 – 322, 1998.

[FA95] S. Frølund and G. Agha. Abstracting Interactions Based on Message Sets. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, pages 107–124. Springer-Verlag, Berlin, 1995.

[FGL+96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96)*, pages 406–421. Springer-Verlag, 1996.

[FHW99] C. J. Fidge, I. J. Hayes, and G. Watson. The deadline command. In *IEE Proceedings - Software*, volume 2, pages 104–111, April 1999.

[FK95] Wan Fokkink and Steven Klusener. An effective axiomatization for real time ACP. *Information and Computation*, 122(2):286–299, 1 November 1995.

[FK98] S. Frølund and J. Koistinen. QML: A language for quality of service specication. February 1998.

[Flo67] R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Mathematical aspects of computer science: Proc. American Mathematics Soc. symposia*, volume 19, pages 19–31, Providence RI, 1967. American Mathematical Society.

[FM91] J. Fiadeiro and T. Maibaum. Temporal reasoning over deontic specifications. *Journal of Logic and Computation*, 1(3):357–395, May 1991.

[FMQ94] G. Ferrari, U. Montanari, and P. Quaglia. A pi-calculus with explicit substitution: the late semantics. In *MFCS*. Springer-Verlag, 1994.

[FNLL96] A. Février, E. Najm, G. Leduc, and L. Léonard. Compositional specification of odp binding objects. In *In Proceedings of the 6th IFIP/ICCC Conference on Information Network and Data Communication, INDC'96*, Trondheim, Norway, June 1996.

[FNS97] Arnaud Février, Elie Najm, and Jean-Bernard Stefani. Contracts for odp. In *Transformation-Based Reactive Systems Development, 4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software, ARTS'97*, volume 1231 of *Lecture Notes in Computer Science*, pages 216–232. Springer, 1997.

[Fu95] C. Fidge and J. Žic. A. simple, expressive real-time CCS. In *Proc. 2nd Australasian Conf. on Parallel & Real-Time Systems*, pages 365–372, 1995.

[Gay99] Simon Gay. Some type systems for the pi calculus, 1999. Manuscript.

[GBSS98] R. Grosu, M. Broy, B. Selic, and G. Stefanescu. Towards a calculus for UML-RT specifications, 1998.

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[Gie00] H. Giese. Contract-based component system design. In J. Ralph and H. Sprague, editors, *33 Annual Hawaii Intern. Conf. on System Sciences (HICSS-33)*, Maui, USA, 2000.

[GL92] J. C. Godskesen and K. G. Larsen. Real–time calculi and expansion theorems. In Rudrapatna Shyamasundar, editor, *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, volume 652 of *LNCS*, pages 302–315, Berlin, Germany, December 1992. Springer.

[GRS95] Roberto Gorrieri, Marco Roccetti, and Enrico Stancampiano. A theory of processes with durational actions. *Theoretical Computer Science*, 140(1):73–94, 20 March 1995.

[GSSAL94] R. Gawlick, R. Segala, J. Soegaard-Andersen, and N. Lynch. Liveness in timed and untimed systems. *Lecture Notes in Computer Science*, 820:166–184, 1994.

[GVR02] Simon Gay, Vasco T. Vasconcelos, and António Ravara. Session types for inter-process communication. October 2002.

[Hew77] Carl E. Hewitt. Viewing control structures as pattern of passing messages. *Artificial Intelligence: An International Journal*, 8(3):323–364, June 1977.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices*, pages 169–180, October 1990. Published as Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices, volume 25, number 10.

[HL95] M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138(2):353–389, 20 February 1995.

[HLR92] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous dataflow language LUSTRE. *IEEE Transactions on Software Engineering*, 18(9):785–793, September 1992.

[HMP91] Tom Henzinger, Zohar Manna, and Amir Pnueli. Temporal proof methodologies for real-time systems. In ACM, editor, *POPL '91. Proceedings of the eighteenth annual ACM symposium on Principles of programming languages, January 21–23, 1991, Orlando, FL*, pages 353–366, New York, NY, USA, 1991. ACM Press.

[HNSY92] T.A. Henzinger, Z. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Logic in Computer Science*, 1992.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs (NJ), USA, 1985. Traduction française : [Hoa87].

[Hoa87] C. A. R. Hoare. *Processus séquentiels communiquants.* Masson, Paris, 1987. Traduction française de [Hoa85].

[Hol92] Ian M. Holland. Specifying Reusable Components Using Contracts. In O. Lehrmann Madsen, editor, *Proceedings of the ECOOP '92 European Conference on Object-oriented Programming*, LNCS 615, pages 287–308, Utrecht, The Netherlands, July 1992. Springer-Verlag.

[Hoo94] Jozef Hooman. Extending hoare logic to real-time. *Formal Aspects of Computing*, 6(6A):801–826, 1994.

[Hoo98] J. Hooman. Compositional verification of real-time applications. In W-P. de Roever, H. Langmaack, and A. Pnueli, editors, *Proceedings of COMPOS'97*, number 1536 in LNCS, pages 276–300. Springer-Verlag, 1998.

[HPR97] N. Halbwachs, Y. E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.

[HQRT98] T. A. Henzinger, S. Qadeer, S. K. Rajamani, and S. Tasiran. An assume-guarantee rule for checking simulation. *Lecture Notes in Computer Science*, 1522:421–433, 1998.

[HR95] Matthew Hennessy and Tim Regan. A process algebra for timed systems. *Information and Computation*, 117(2):221–239, March 1995.

[HR02] Hennessy and Riely. Resource access control in systems of mobile agents. *INFCTRL: Information and Computation (formerly Information and Control)*, 173, 2002.

[HW89] Jozef Hooman and Jennifer Widom. A temporal-logic based compositional proof system for real-time message passing. *Parallel Architectures and Languages in Europe, PARLE '89, Eindhoven, Netherlands*, pages 424–441, June 1989.

[HX91] He Jifeng and Xu Qiwen. A theory of state-based parallel programming by refinement. In *Proc. 1991 Refinement Workshop, Cambridge*, 1991.

[HYL92] U. Holmer, W. Yi, and K. Larsen. Deciding properties of regular real timed processes. In Kim G. Larsen and Arne Skou, editors, *Proceedings of Computer Aided Verification (CAV '91)*, volume 575 of *LNCS*, pages 443–453, Berlin, Germany, July 1992. Springer.

[IK01] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the Pi-calculus. *ACM SIGPLAN Notices*, 36(3):128–141, March 2001.

[JLS00] Henrik Ejersbo Jensen, Kim Guldstrand Larsen, and Arne Skou. Scaling up uppaal automatic verification of real-time systems using compositionality and abstraction. In *FTRTFT*, pages 19–30, 2000.

[Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, October 1983.

[JSV93] Alan S. A. Jeffrey, Steve A. Schneider, and Frits W. Vaandrager. A comparison of additivity axioms in timed transition systems. In *87*, page 19. Centrum voor Wiskunde en Informatica (CWI), ISSN 0169-118X, December 31 1993.

[JT95] B. Jonsson and Y.-K. Tsay. Assumption/guarantee specifications in linear-time temporal logic. *Lecture Notes in Computer Science*, 915:262–264, 1995.

[KKLS00] Idit Keidar, Roger Khazan, Nancy A. Lynch, and Alexander A. Shvartsman. An inheritance-based technique for building simulation proofs incrementally. In *International Conference on Software Engineering*, pages 478–487, 2000.

[Kön00] Barbara König. Analysing input/output-capabilities of mobile processes with a generic type system (extended version). Technical Report Technical Report TUM-I0009, Technische Universitat Munchen, 2000.

[Kob00] Naoki Kobayashi. Type systems for concurrent processes: From deadlock-freedom to livelock-freedom, time-boundedness. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. D. Mosses, and T. Ito, editors, *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics, Proceedings of the International IFIP Conference TCS 2000 (Sendai, Japan)*, volume 1872 of *LNCS*, pages 365–389. IFIP, Springer, August 2000.

[Kob02] Naoki Kobayashi. A type system for lock-free processes. *Information and Computation*, 177(2):122–159, September 2002.

[KPT99] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the Pi-Calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.

[LBS+98] Joseph P. Loyall, David D. Bakken, Richard E. Schantz, John A. Zinky, David A. Karr, Rodrigo Vanegas, and Kenneth R. Anderson. QoS aspect languages and their runtime integration. In *Proceedings of the 4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR)*, volume 1511, Berlin, Heidelberg, New York, Tokyo, 1998. Springer-Verlag.

[Lee88] Ronald M. Lee. A logic model for electronic contracting. *Decision Support Systems*, 4(1):27–44, 1988.

[Leu95] S. Leue. Specifying real-time requirements for sdl specifications - a temporal logic-based approach. In *Proceedings of the Fifteenth International Symposium on Protocol Specification, Testing, and Verification PSTV'95*. Chapmann & Hall, 1995.

[Lin94] Huimin Lin. Symbolic Bisimulations and Proof Systems for the Pi-Calculus. Technical Report 94:07, 1994.

[Lin98] Huimin Lin. Complete proof systems for observation congruences in finite-control pi-calculus. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *25th Colloquium on Automata, Languages and Programming (ICALP) (Aalborg, Denmark)*, volume 1443 of *LNCS*, pages 443–454. Springer, July 1998.

[Lin03] Huimin Lin. Complete inference systems for weak bisimulation equivalences in the *pi*-calculus. *Information and Computation*, 180(1):1–29, 2003.

[LL92] Guy Leduc and Luc Leonard. A timed LOTOS supporting a dense time domain and including new timed operators. In *FORTE*, pages 87–102, 1992.

[LL98] L. Léonard and G. Leduc. A formal definition of time in LOTOS. In *Formal Aspects of Computing*, volume 10, pages 248–266, 1998.

[LMB+96] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul T. Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1280–1297, 1996.

[LNW02] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Invited paper, Journal of Circuits, Systems, and Computers*, November 20, 2002.

[LPS81] Daniel J. Lehmann, Amir Pnueli, and Jonathan Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In Shimon Even and Oded Kariv, editors, *Automata, Languages and Programming, 8th*

Colloquium, volume 115 of *Lecture Notes in Computer Science*, pages 264–277, Acre (Akko), Israel, 13–17 July 1981. Springer-Verlag.

[LS84] Leslie Lamport and Fred B. Schneider. The Hoare Logic of CSP, and all that. *ACM Transactions on Programming Languages and Systems*, 6(2):281–296, April 1984.

[LS91] Simon S. Lam and A. Udaya Shankar. Understanding interfaces. In *Proceedings Fourth International Conference on Formal Description Techniques (FORTE'91)*, Sydney, Australia, November 1991.

[LSZB98] J. P. Loyall, R. E. Schantz, J. A. Zinky, and D. E. Bakken. Specifying and measuring quality of service in distributed object systems. In *Proceedings of ISORC'98*, Kyoto, Japan, April 1998.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 137–151. ACM Press, 1987.

[LW94] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[LX01] Edward A. Lee and Yuhong Xiong. System-level types for component-based design. In *First Workshop on Embedded Software, EMSOFT2001*, October 2001.

[LY00] Lin and Yi. A complete axiomatisation for timed automata. *FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, 20, 2000.

[LY02] H. Lin and Wang Yi. Axiomatixing timed automata. *Acta Informatica*, 38:277–305, 2002.

[MC81] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, July 1981.

[McM99] Kenneth L. McMillan. Circular compositional reasoning about liveness. In *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference (CHARME'99)*, volume 1703 of *LNCS*, pages 342–345. Springer, 1999.

[Mey89] B. Meyer. *Object-oriented software construction.* Prentice Hall, New York, 1989.

[Mey91] B. Meyer. *Eiffel: The Language.* Prentice Hall, 1991.

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[MF76] P. Merlin and D. J. Faber. Recoverability of communication protocols. *IEEE Transactions on Communication*, 24(9):1036–1043, 1976.

[MG95] Marino Miculan and Fabio Gadducci. Modal μ-types for processes. In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 221–231, San Diego, California, 26–29 June 1995. IEEE Computer Society Press.

[Mic95] Sun Microsystems. The java virtual machine specification. Technical report, Sun Microsystems, Mountain View, California, august 1995.

[Mil83] Robin Milner. *A calculus of communicating systems*, volume 158 of *Lecture Notes in Computer Science*. Springer-Verlag, New York-Berlin-Heidelberg, 1983.

[Mil84] Robin Milner. A complete inference system for a class of regular behaviours. *Journal of Computer and System Sciences*, 28:439–466, 1984.

[Mil89a] Robin Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.

[Mil89b] Robin Milner. A complete axiomatisation for observational congruence of finite-state behaviours. *Information and Computation (formerly Information and Control)*, 81, 1989.

[Mil92] Robin Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992. Previous version as Rapport de Recherche 1154, INRIA Sophia-Antipolis, 1990, and in *Proceedings of ICALP '91*, LNCS 443.

[Mil93] Robin Milner. The polyadic pi-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.

[MLM94] Sape J. Mullender, Ian M. Leslie, and Derek McAuley. Operating system support for distributed multimedia. In *USENIX Summer*, pages 209–219, 1994.

[MNCK99] S. Mitchell, H. Naguib, G. Coulouris, and T. Kindberg. A qos support framework for dynamically reconfigurable multimedia applications. In *DAIS'99*, June 1999.

[Mol90] C. Tofts F. Moller. A temporal calculus of communicating systems. In *Concur'90*, LNCS, pages 401–415. Springer-Verlag, 1990.

[MP91] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer-Verlag, 1991.

[MP95] Ugo Montanari and Marco Pistore. Checking bisimilarity for finitary π-calculus. In Insup Lee and Scott A. Smolka, editors, *CONCUR '95: Concurrency Theory, 6th International Conference*, volume 962 of *Lecture Notes in Computer Science*, pages 42–56, Philadelphia, Pennsylvania, 21–24 August 1995. Springer-Verlag.

[MP98] Ugo Montanari and Marco Pistore. History-dependent automata. Technical Report TR-98-11, Dipartimento di Informatica, October 5 1998.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Information and Computation*, 100(1):41–77, 1992.

[MPW93] Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114(1):149–171, 1993.

[MT91] Faron Moller and Chris Tofts. Relating processes with respect to speed. In J. C. M. Baeten and J. F. Groote, editors, *CONCUR '91: 2nd International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*, pages 424–438, Amsterdam, The Netherlands, 26–29 August 1991. Springer-Verlag.

[MW93] J.-J.Ch. Meyer and R.J. Wieringa, editors. *Deontic Logic in Computer Science: Normative System Specification*, New York, 1993. J. Wiley.

[NA99] Brian Nielsen and Gul Agha. Towards reusable real-time objects. In *Annals of Software Engineering*, volume 7, pages 257–282, 1999.

[NH83] Rocco De Nicola and Matthew C. B. Hennessy. Testing equivalence for processes. In Josep Díaz, editor, *Automata, Languages and Programming, 10th Colloquium*, volume 154 of *Lecture Notes in Computer Science*, pages 548–560, Barcelona, Spain, 18–22 July 1983. Springer-Verlag.

[Nie95] O. Nierstrasz. Regular types for active objects. In *Object-Oriented Software Composition*. Prentice Hall, 1995.

[NL96] G. Necula and P. Lee. Proof-carrying code. Technical report cmu-cs-96165, School of Computer Science, Carnegie Mellon University, September 1996.

[NN97] Elie Najm and Abdelkrim Nimour. A calculus of object bindings. In Howard Bowman and John Derrick, editors, *Proceedings of FMOODS'97*. Chapman & Hall, 1997.

[NNS99a] Elie Najm, Abdekrim Nimour, and J-B Stefani. Infinite types for distributed objects interfaces. In *Proc. of IFIP conf. FMOODS'99*. Kluwer, Feb 1999.

[NNS99b] Elie Najm, Abdelkrim Nimour, and Jean-Bernard Stefani. Guaranteeing liveness in an object calculus through behavioral typing. In *FORTE/PSTV'99*. Kluwer Academic Publishers, 1999.

[NS92] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 526–548. Springer-Verlag, 1992.

[NSY93] Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. From ATP to timed graphs and hybrid systems. *Acta Informatica*, 30(2):181–202, 1993.

[NT00] Kedar S. Namjoshi and Richard J. Trefler. On the completeness of compositional reasoning. In *Proceedings of the 12th Int. Conference on Computer Aided Verification (CAV2000)*, number 1855, pages 139–153. Springer-Verlag, 2000.

[ODP95] Open distributed processing reference model, parts 1,2,3,4. ISO/IEC IS 10746-1..4 or ITU-T X901..4, 1995.

[OG76] Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976. Papers from the Fifth ACM Symposium on Operating Systems Principles (Univ. Texas, Austin, Tex., 1975).

[Ost89] J. Ostroff. *Temporal Logic for Real-Time Systems*. Advanced Software Development Series. Research Studies Press Ltd., 1989.

[Par85] Joachim Parrow. *Fairness Properties in Process Algebra*. PhD thesis, Uppsala, 1985.

[PJ91] P. K. Pandya and M. Joseph. P-A logic - A compositional proof system for distributed programs. *Distributed Computing*, 5(1):37–54, 1991.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[Pnu84] A. Pnueli. In transition from global to modular temporal reasoning about programs. In Krzystof R. Apt, editor, *Logics and Model of Concurrent Systems*, volume 13 of *NATO ASI*, pages 123–144. Springer-Verlag, October 1984.

[PP99] Franz Puntigam and Christof Peter. Changeable interfaces and promised messages for concurrent components. In *Proceedings of the ACM Symposium on Applied Computing (SAC'99)*, San Antonio, Texas, USA, 1999.

[PS93] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Proceedings 8th IEEE Logics in Computer Science*, pages 376–385, Montreal, Canada, 1993.

[PT00] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[Pun97] Franz Puntigam. Coordination requirements expressed in types for active objects. In *European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 367–387. Springer-Verlag, June 1997.

[Pun99] Franz Puntigam. Non-regular process types. In P. Amestoy et al., editors, *Proceedings of the 5th European Conference on Parallel Processing (Euro-Par'99)*, number 1685, Toulouse, France, 1999. Springer-Verlag.

[Qua99] P. Quaglia. The pi-calculus: Notes on labelled semantics. *Bulletin of the European Association for Theoretical Computer Science*, 68:104–, 1999.

[R. 91] R. Alur. *Techniques for Automatic Verification of Real-Time Systems.* PhD thesis, Stanford University, Stanford, 1991.

[RA95] Shangping Ren and Gul A. Agha. RTsynchronizer: language support for real-time specifications in distributed systems. *ACM SIGPLAN Notices*, 30(11):50–59, November 1995.

[Ram74] C. Ramchandani. ANALYSIS OF ASYNCHRONOUS CONCURRENT SYSTEMS BY TIMED PETRI NETS. Technical Report MIT/LCS/TR-120, 1974.

[RE00] H.O. Rafaelsen and F. Eliassen. Trading and negotiating stream bindings. In *in Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, New-York, 2000.

[RR01] Sriram K. Rajamani and Jakob Rehof. A behavioral module system for the pi-calculus. *Lecture Notes in Computer Science*, 2126:375–377, 2001.

[RRV99] António Ravara, Pedro Resende, and Vasco T. Vasconcelos. An algebra of behavioural types. Preprint 26–99, ISTDM, September 1999.

[RS] Jim Rumbaugh and Bran Selic. Using UML for modeling complex real-time systems. Rational Software Corp. and ObjectTime Limited.

[RT91] R. Alur and T.A. Henzinger. Logics and Models of Real-Time: A Survey. In *Real Time: Theory in Practice*, volume 600 of *LNCS*, pages 74–106. Springer-Verlag, 1991.

[RV97] A. Ravavra and V.T. Vasconcelos. Behavioral types for a calculus of concurrent objects. In *Euro-Par'97*. Springer-Verlag, 1997.

[RV00] A. Ravara and V. Vasconcelos. Typing non-uniform concurrent objects. In C. Palamidessi, editor, *CONCUR'00*, volume 1877 of *Lecture Notes in Computer Science*, pages 474–488. Springer-Verlag, 2000.

[San93] Davide Sangiorgi. A theory of bisimulation for the π-calculus. In Eike Best, editor, *CONCUR '93: 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 127–142, Hildesheim, Germany, 23–26August 1993. Springer-Verlag.

[San96] Davide Sangiorgi. Locality and non-interleaving semantics in calculi for mobile processes. *Theoretical Computer Science*, 155:39–83, 1996.

[San99] Davide Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 221(1–2):457–493, 1999.

[SC00] P. N. M. Sampaio and J. P. Courtiat. A formal approach for the presentation of interactive multimedia documents. In *Proceedings of the 8th International ACM Conference on Multimedia (Multimedia-00)*, pages 432–434, N. Y., October 30–November 04 2000. ACM Press.

[Sch95] Steve Schneider. An operational semantics for timed CSP. *Information and Computation*, 116(2):193–213, 1 February 1995.

[Sch00] D. C. Schmidt. Real time CORBA with TAO (the ACE ORB). Technical report, Washington University in Saint Louis, http://www.cs.wustl.edu/schmidt/TAO.html, 2000.

[Seg92] R. Segala. A Process Algebraic View of I/O Automata. Technical Report MIT/LCS/TR-557, 1992.

[Seg97] Roberto Segala. Quiescence, fairness, testing, and the notion of implementation. *Information and Computation*, 138(2):194–210, 1 November 1997.

[Sel96] Bran Selic. Real-time object-oriented modeling (ROOM). In *IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, pages 214–219, Washington - Brussels - Tokyo, June 1996. IEEE Computer Society Press.

[SHG89] R. K. Shyamasundar, J. Hooman, and R. Gerth. Reasoning of real-time distributed programming languages. In *Proceedings: Fifth International Workshop on Software Specification and Design*, pages 91–99, 1989.

[Sig99] M. Sighireanu. Contribution at the definition and implementation of e-lotos, 1999.

[SM73] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*. ACM, 1973.

[Sol97] F. Solms. Unified modeling language for real-time systems design, 1997.

[SRRS01] Sekar, Ramakrishnan, Ramakrishnan, and Smolka. Model-carrying code (MCC): A new paradigm for mobile-code security. In *WNSP: New Security Paradigms Workshop*. ACM Press, 2001.

[SSdSC98] C. A. S. Santos, L. F. G. Soares, G. L. de Souza, and J. P. Courtiat. Design methodology and formal validation of hypermedia documents. In *Proceedings of the 6th ACM International Conference on Multimedia (Multimedia-98)*, pages 39–48, N.Y., September 12–16 1998. ACM Press.

[ST92] Ichiro Satoh and Mario Tokoro. A formalism for real-time concurrent object-oriented computing. *ACM SIGPLAN Notices*, 27(10):315–326, October 1992.

[Sta85] E. W. Stark. A proof technique for rely/guarantee properties. In S. N. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 206 of *LNCS*, pages 369–391, New Delhi, 1985. Springer-Verlag.

[Sti88] Colin Stirling. A generalization of Owicki-Gries's Hoare logic for a concurrent while language. *Theoretical Computer Science*, 58(1-3):347–359, June 1988.

[Stø91] Ketil Stølen. A method for the development of totally correct shared-state parallel programs. In J. C. M. Baeten and J. F. Groote, editors, *CONCUR '91: 2nd International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*, pages 510–525, Amsterdam, The Netherlands, 26–29 August 1991. Springer-Verlag.

[SW01] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[SY96] Joseph Sifakis and Sergio Yovine. Compositional specification of timed systems (extended abstract). In *13th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1046 of *lncs*, pages 347–359, Grenoble, France, 22–24 February 1996. Springer.

[TJ94] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, June 1994.

[TMM88] Mark Tuttle, Michael Meritt, and Francesmary Modugno. Time constrained automata. MIT/LCS, November 1988.

[Tri99] Stavros Tripakis. Verifying progress in timed systems. *Lecture Notes in Computer Science*, 1601:299–314, 1999.

[TY01] Stavros Tripakis and Sergio Yovine. Analysis of timed systems using time-abstracting bisimulations. *Formal Methods in System Design*, 18(1):25–68, 2001.

[TZH02] David Teller, Pascal Zimmer, and Daniel Hirschkoff. Using ambients to control resources. In *CONCUR 2002, 19th International Conference on Concurrency Theory*, number 2421 in LNCS, pages 288–303. Springer, 2002.

[Čer92] Kārlis Čerāns. Decidability of bisimulation equivalences for parallel timer processes. In *CAV'92*, volume 663 of *Lecture Notes in Computer Science*, Berlin, 1992. Springer Verlag.

[Vaa91] Frits W. Vaandrager. On the relationship between process algebra and input/output automata. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 387–398, Amsterdam, The Netherlands, 15–18 July 1991. IEEE Computer Society Press.

[VB98] Vasco T. Vasconcelos and Rui Bastos. Core-TyCO, the language definition, version 0.1. DI/FCUL TR 98–3, DIFCUL, March 1998.

[vG01] R. van Glabbeek. *Handbook of Process Algebra*, chapter The Linear Time – Branching Time Spectrum I: The Semantics of Concrete, Sequential Processes. Elsevier Science, Amsterdam, The Netherlands, 2001.

[VL92] F. Vaandrager and N. Lynch. Action transducers and timed automata. *Lecture Notes in Computer Science*, 630:436–454, 1992.

[VM94] Böjrn Victor and Faron Moller. The mobility workbench. A tool for the pi-calculus. LFCS report ECS-LFCS-94-285, Department of Computer Science, University of Edinburgh, JCMB, The Kings Buildings, Mayfield Road, Edinburgh, 1994.

[von51] G. H. von Wright. Deontic logic. *Mind*, 60(237):1–15, 1951.

[VR99] Vasco T. Vasconcelos and António Ravara. Communication errors in the π-calculus are undecidable. *Information Processing Letters*, 71(5–6):229–233, September 1999.

[VVR02] Antonio Vallecillo, Vasco T. Vasconcelos, and António Ravara. Typing the behavior of objects and components using session types. In *1st International Workshop on Foundations of Coordination Languages and Software Architectures (Foclasa 2002)*, Electronic Notes in Theoretical Computer Science. Elsevier, August 2002.

[WG02] Peter Wegner and Dina Goldin. Computation beyond turing machines. *Communications of the ACM*, 2002. To be published.

[WJGO98] I. Wakeman, A. Jeffrey, R. Graves, and T. Owen. Designing a programming language for active networks. Technical report, University of Sussex, 1998.

[WN95] G. Winskel and M. Nielsen. Models for concurrency. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, 1995.

[WZ88] Peter Wegner and Stanley B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In S. Gjessing and K. Nygaard, editors, *ECOOP '88, European Conference on Object-Oriented Programming, Oslo, Norway*, volume 322 of *Lecture Notes in Computer Science*, pages 55–77, New York, NY, August 1988. Springer-Verlag.

[XCC94] Qiwen Xu, Antonio Cau, and Pierre Collette. On unifying assumption-commitment style proof rules for concurrency. In *International Conference on Concurrency Theory*, pages 267–282, 1994.

[XdRH97] Qiwen Xu, Willem P. de Roever, and Jifeng He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

[YH00] Nobuko Yoshida and Matthew Hennessy. Assigning types to processes. In *Logic in Computer Science*, pages 334–345, 2000.

[Yi91] Wang Yi. Ccs + time = an interleaving model for real time systems. In *Automata, Languages and Programming (ICALP'91)*, volume 510 of *LNCS*, pages 217–228. Springer-Verlag, 1991.

[Yos96] Nobuko Yoshida. Graph types for monadic mobile processes. In *Foundations of Software Technology and Theoretical Computer Science*, pages 371–386, 1996.

[Yos02] Nobuko Yoshida. Type-based liveness guarantee in the presence of nontermination and nondeterminism. MCS 2002-20, University of Leicester, April 2002.

[ZdRvEB85] Job Zwiers, Willem P. de Roever, and Peter van Emde Boas. Compositionality and concurrent networks: Soundness and completeness of a proofsystem. In Wilfried Brauer, editor, *Automata, Languages and Programming, 12th Colloquium*, volume 194 of *Lecture Notes in Computer Science*, pages 509–519, Nafplion, Greece, 15–19 July 1985. Springer-Verlag.

[Zwi89] J. Zwiers. *Compositionality, concurrency and partial correctness.* Springer-Verlag, New York, 1989.