Langages dédiés au développement de services de communications

Nicolas Palix

N° d'ordre: 3623

THÈSE

présentée à

L'UNIVERSITÉ BORDEAUX 1
École Doctorale de Mathématiques et Informatique

par **Nicolas Palix**

pour obtenir le grade de

DOCTEUR
Spécialité: INFORMATIQUE

Langages dédiés au développement de services de communications

Thèse dirigée par Charles Consel

Soutenue le : 17 septembre 2008

Devant la commission d'examen présidée par Daniel Hagimont et formée de :

MM. :	Pierre	Cointe	Professeur à l'École des Mines de Nantes	Rapporteurs
	Daniel	Hagimont	Professeur à l'ENSEEIHT, INP de Toulouse	
MM. :	Renaud	Marlet	Chargé de recherche à l'INRIA, Bordeaux	Examinateurs
	Charles	Consel	Professeur à l'ENSEIRB, Bordeaux	
M. :	Toufik	Ahmed	Maître de conférence à l'ENSEIRB, Bordeaux	Invité

Université Bordeaux 1 – Les Sciences et les Technologies au service de l'Homme et de l'environnement

Thèse réalisée au sein de l'Équipe-Projet INRIA Phoenix

INRIA Bordeaux – Sud-Ouest
Bâtiment A 29
351 cours de la libération
F-33405 Talence Cedex

LaBRI
Unité Mixte de Recherche CNRS (UMR 5800)
351 cours de la libération
F-33405 Talence cedex

à Aurore et Aymie,

Remerciements

Je tiens tout d'abord à remercier les membres de mon jury :
- Daniel Hagimont et Pierre Cointe qui ont assumé la charge de rapporteur, leurs avis et commentaires furent une aide précieuse dans les dernières semaines et les derniers jours. Je remercie plus particulièrement Daniel Hagimont qui m'a également fait l'honneur de présider ce jury,
- Renaud Marlet dont la lecture minutieuse de cette thèse a permis de corriger et de préciser de nombreux détails, y compris dans les annexes,
- Toufik Ahmed d'avoir lu en détail ce document.

Je tiens ensuite à remercier mon directeur de thèse, Charles Consel, qui m'a accueilli au sein de l'équipe projet INRIA Phoenix. Grâce à ses remarques et ses conseils, j'ai pu mener à terme ces travaux.

Laurent Réveillère et Julia Lawall ont également participé aux succès de ces travaux. Leur aide et leur savoir-faire m'ont indéniablement permis de progresser.

Laurent Burgy, avec qui j'ai partagé quatre bureaux différents au cours de ces quatres années, m'a permis de surmonter les instants de doutes. Je n'oublierai pas non plus nos discussions autour d'un café et son aide précieuse tout au long de ma thèse, et plus particulièrement pendant la rédaction de ce manuscrit. Il a en effet été le premier lecteur de ce manuscript. J'avais un collégue, j'ai trouvé un ami.

Wilfried Jouve, *a.k.a* Wiwi, qui a également partagé mon bureau. Sans rancune, j'espère ! Je me souviendrai de sa compagnie appréciée lors de nos voyages en Allemagne et en Hollande. Même si son niveau d'allemand est ... disons scolaire, nous sommes malgré tout arriver à bon port. Enfin, si vous pouvez lire ces lignes, c'est grâce à lui et je l'en remercie.

Je remercie également les autres membres de l'équipe avec qui j'ai partagé des moments agréables autour d'un café ou d'un baby-foot.

Je remercie affectueusement mes parents, Monique et Michel, pour m'avoir permis des études aussi longue. Leur soutien indéfectible a rendu possible ma réussite scolaire. Je remercie également ma sœur, Anne-Gaëlle, et lui souhaite de réussir dans sa nouvelle vie en *freelance*. Je remercie mes beaux-parents, Anne-Marie et Hubert, dont j'ai respectivement apprécié la cuisine et les conseils. Je n'aurais peut-être jamais commencé de thèse sans les encouragements de mon beau-père.

Je remercie enfin ma femme, Aurore, et notre fille, Aymie. Le soutien d'Aurore dans les instants de doute, et plus particulièrement lors de la rédaction est inestimable. J'espère pouvoir lui apporter un soutien aussi fort tout au long de notre vie. Je lui dois aussi notre fille, Aymie. Ses nombreux sourires et sa tendresse m'ont permis des instants de repos salutaire pendant la rédaction. Enfant modèle, j'ai pu avoir des nuits presque normale. Que la vie t'apporte santé, bonheur et joie Aymie, garde ce sourire qui te va si bien.

Nicolas Palix, Copenhague *(København)*, le 1er novembre 2008

Langages dédiés au développement de services de communications

Résumé

Les services de téléphonie IP exploitent des ressources réseaux pour automatiser le traitement des stimuli de communication. Cependant, l'ajout de services rend vulnérable un système de téléphonie et peut interrompre son fonctionnement nominal. Pour préserver un système de téléphonie, il faut garantir certaines propriétés de fonctionnement des services déployés. Il existe des solutions de développement de services garantissant des propriétés de fonctionnement mais leur expressivité est réduite, limitant grandement leur utilisation.

Cette thèse propose une approche fondée sur le concept des langages dédiés pour développer des services de communications. Les langages dédiés *(Domain-Specific Languages)* permettent de répondre au besoin de développement dans un domaine particulier. Ils introduisent une expressivité et un degré de vérification adaptés et spécifiques à un domaine. Les propriétés critiques d'un domaine sont vérifiées statiquement par le compilateur du langage.

Deux nouveaux langages dédiés au domaine des communications ont été développés : SPL (Session Programming Language) et Pantaxou. Le langage SPL sert à traiter des messages de signalisation pour la téléphonie IP. SPL offre des abstractions spécifiques pour garantir les propriétés critiques de la téléphonie IP. Le langage Pantaxou généralise la notion de services de communications introduite par SPL en permettant la coordination d'entités communicantes. Ce langage se décompose en deux parties : la première consiste à décrire un environnement d'entités communicantes, tandis que la seconde permet d'exprimer des logiques de coordination pour ces entités. Ces logiques de coordination doivent respecter les contraintes exprimées par la description d'environnement.

Les contributions de cette thèse sont les suivantes :

- Nous avons effectué une analyse des services de communications. Cette analyse se concentre sur les services de communications fondés sur le protocole SIP. Les approches existantes pour le développement de services de communications fondées sur SIP sont notamment présentées.
- Nous avons conçu un langage, nommé SPL, permettant de développer des services de routage. Diverses analyses pour ce langage garantissent des propriétés critiques du protocole SIP sous-jacent.
- Nous avons généralisé le développement de services de communications. Nous avons dans ce cadre conçu un langage, nommé Pantaxou, qui propose une conception de services de coordination en deux étapes permettant des vérifications en amont du processus de développement.
- La conception et l'implémentation de ces langages améliorent le processus de développement des services de communications et notamment leur fiabilité.

L'approche fondée sur l'utilisation des langages dédiés que nous proposons dans cette thèse ouvre, au delà des services de téléphonie, de nouvelles perspectives quant au développement de systèmes distribués comme les systèmes ubiquitaires.

Mots clés

Langages dédiés, services de communications, téléphonie IP, génie logiciel

Domain-Specific Languages for Developing Communication Services

Abstract

IP telephony services use network resources to automate communication stimuli processing. However, deploying services on a telephony system leads to safety issues, and programmers need to ensure some safety properties on their services. Several approaches allowing service development have emerged. Some of them ensure safety properties by restricting expressiveness, limiting the scope of usage.

This thesis proposes a new approach that relies on *domain-specific languages* (DSL) to develop communication services. A domain-specific language is a programming language dedicated to a particular domain. Their expressiveness and verifiability are adapted and specific to a domain. A compiler statically checks critical domain properties.

Two new DSLs have been designed for communication services, namely SPL (Session Processing Language) and Pantaxou. The SPL language allows developers to specify routing logic that processes IP telephony signaling. SPL offers abstractions specific to message processing that guarantee the safety properties of IP telephony. The Pantaxou language generalizes the communication service concept introduced by SPL allowing the coordination of communicating entities. The language consists of two parts: the first one enables the description of an environment of communicating entities; the second one expresses the coordination scenarios between these entities. These scenarios must ensure some constrains with respect to the environment description. Two compilers ensure the environment consistency and the scenarios consistency regarding the environment.

The contributions of this thesis are as follows:

- We carry out an analysis of communication services. This analysis focuses on communication services based on the SIP protocol. Existing approaches for developing SIP-based communication services are presented.
- We present the design and implementation of a language, named SPL, for programming routing services. Various analyses for this language ensure critical properties of the SIP protocol.
- We generalize the service development to entity services. We have designed a language, named Pantaxou, that introduces a two-step process for the development of coordination services, enabling early verification during the development process.
- The design and implementation of these languages improve the development process of communication services and especially their safety.

Beyond the domain of telephony services, our DSL-based approach opens up new possibilities for the development of distributed services in the field of ubiquitous computing.

Key words

Domain-Specific Languages, Communication Services, IP Telephony, Software Engineering

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

La démocratisation d'Internet et la récente convergence des réseaux de télécommunications et des réseaux informatiques ont permis un essor spectaculaire des communications de tous types. La convergence de réseaux de toutes natures, vers un unique réseau informatique, s'est accompagnée de la création de protocoles pour les communications audio et vidéo, historiquement dévolues aux réseaux de télécommunications. Le protocole SIP occupe aujourd'hui une place prépondérante parmi ces protocoles ; il est déployé par les principaux opérateurs téléphoniques.

Pour répondre à un volume de communications de plus en plus important, l'automatisation de leur traitement devient une priorité. Pour répondre à ce besoin, l'émergence du protocole SIP s'est accompagnée de la conception d'approches pour le développement de services de communications. Ces approches profitent de la convergence des réseaux et permettent de fournir de nouveaux services aux utilisateurs.

Les services de communications peuvent être catégorisés selon deux classes : les services de routage et les services entités. Les services de routage interceptent les stimuli de communications entre deux entités communicantes. Ils modifient éventuellement ces stimuli afin de rediriger les appels par exemple. Les services entités sont des entités communicantes. Ces services reçoivent des stimuli de communication et les traitent. Mais contrairement aux services de routage, ils peuvent aussi générer de tels stimuli. Par exemple, un service de rappels automatiques met en relation deux interlocuteurs dès qu'ils sont disponibles.

Les approches existantes pour développer des services sont de deux ordres : les approches fondées sur les langages généralistes et les approches fondées sur des langages dédiés. Un langage dédié est un langage de programmation spécialisé pour un domaine ou une classe de problèmes. La spécialisation d'un langage à un domaine se matérialise par des abstractions et des notations spécifiques à ce domaine. L'introduction de ces abstractions et notations s'accompagne généralement de restrictions permettant de garantir des propriétés critiques du domaine. Cependant aucune solution n'est actuellement appropriée. D'un coté, les solutions généralistes sont complexes et nécessitent des compétences multiples notamment en programmation, télécommunications et réseaux. Ces solutions ne permettent pas un développement simple et sûr de services. D'un autre coté, les approches dédiées existantes sont haut niveau mais lorsqu'elles sont expressives, elles ne sont plus sûres. À l'inverse, les approches sûres ne fournissent plus autant d'expressivité, restreignant ainsi les possibilités offertes. Il est donc souhaitable que les développeurs de services de communications puissent concevoir leurs services à l'aide d'une nouvelle approche haut niveau, sûre et expressive.

1.1 Thèse

Ce document propose une approche pour le développement de services de communications. Cette approche repose sur l'introduction de deux langages dédiés permettant le développement de services de routage d'une part, et de services entités d'autre part.

La conception de ces langages dédiés s'est focalisée sur le compromis entre expressivité et sûreté que des abstractions haut niveau permettent. Les propriétés critiques du domaine sont garanties par construction du langage ou analyse des programmes. La faculté d'analyser les programmes pour garantir des propriétés permet de déceler des erreurs ou des incohérences très tôt dans le processus de développement.

Notre approche se fonde sur l'introduction de deux langages dédiés aux services de communications. Un premier langage, nommé SPL, permet le développement de services de routage. Ce langage est fondé sur le protocole de communications SIP, utilisé notamment pour la téléphonie IP. Un second langage, nommé Pantaxou[1], généralise le développement de services de communications aux services entités. Dans les deux cas, les services développés à l'aide de ces langages sont analysés afin de rejeter les programmes erronés ou incohérents. Les services valides sont ensuite déployés dans un environnement d'exécution ad hoc pour y être exécutés.

1.2 Contributions

Les contributions de cette thèse comportent deux volets : l'analyse de domaine des services de communications et deux langages pour le développement de tels services. Les langages proposés concernent d'une part les services de routage avec SPL, et d'autre part les services entités, avec Pantaxou. Nous montrons dans chacun de ces deux cas la faisabilité et l'intérêt d'une approche langage.

Analyse de domaine Nous présentons une analyse des services de communications. Cette analyse s'attache plus spécifiquement aux services de communications fondés sur le protocole SIP. Nous étudions notamment les approches existantes pour le développement de services de communications fondés sur SIP.

Session Processing Langage (SPL) Nous avons conçu le langage SPL pour spécifier des services de routage. Ce langage intègre des abstractions propres aux services de routage et au protocole SIP. Ces abstractions nous permettent de raisonner sur les services développés. Il est ainsi possible de détecter des erreurs ou incohérences propres aux services de routage fondés sur le protocole SIP.

Pantaxou Nous avons généralisé le développement de services aux services entités. Cette généralisation nous a conduit à concevoir le langage Pantaxou dédié à la coordination d'entités communicantes. Pantaxou se compose de deux langages complémentaires : le premier permet de décrire un environnement d'entités communicantes où les services sont déployés ; le second langage s'appuie sur cette description pour faciliter le développement de logiques de coordination. Nous décrivons le processus de développement en Pantaxou qui repose sur un générateur d'intergiciel dédié à un environnement donné et sur un compilateur de logiques de coordination pour cet intergiciel.

[1] Pantaxou vient du grec Πανταχου qui signifie "partout". Pantaxou (ou Pantachou) se prononce [pãtaku]

1.3 Organisation du document

Cette thèse comprend trois parties principales : la présentation du contexte et de la problématique, la description de notre solution fondée sur une approche langage et enfin les apports de notre solution.

Contexte La première partie porte sur l'étude du contexte dans lequel nous nous plaçons. Le chapitre 2 présente les communications et les services de communications dans les réseaux de télécommunications et les réseaux informatiques. Le chapitre 3 s'intéresse plus particulièrement aux communications et aux services dans le cadre de la téléphonie IP. Le chapitre 4 présente l'approche des langages dédiés sur laquelle repose notre solution. Le chapitre 5 décrit les solutions existantes pour le développement de services de communications fondés sur SIP et leurs limitations. Dans le chapitre 6, nous présentons la démarche que nous avons adoptée pour répondre aux problèmes posés par le développement de services de communications. Cette démarche décrit notre approche, de la problématique à la mise en œuvre d'une implémentation.

Approche proposée Dans la deuxième partie, nous décrivons les deux composantes de notre approche. Les chapitres 7 et 8 s'intéressent aux services de routage. Ces chapitres présentent respectivement la conception du langage SPL et la mise en œuvre de ce langage. La conception du langage SPL comprend l'analyse de domaine et la définition du langage. La mise en œuvre du langage présente le processus de développement de service à l'aide de SPL. Le développement d'un service SPL est notamment présenté.

Les chapitres 9 et 10 généralisent l'approche du langage SPL aux services entités. Ces chapitres présentent respectivement la conception du langage Pantaxou et sa mise en œuvre selon la même démarche que le langage SPL.

Bilan Dans la dernière partie de ce document, nous reprenons les apports des approches langages SPL et Pantaxou au chapitre 11. Ces apports sont de deux ordres : génie logiciel et fiabilité. Enfin, le chapitre 12 conclut en récapitulant nos différentes contributions et présente diverses perspectives de recherche.

Première partie

Contexte

Chapitre 2

Services de communications

L'invention du télégraphe électrique puis celle du téléphone ont accompagné la mise en place des premiers réseaux de télécommunications à l'échelle mondiale. Ces réseaux, d'abord analogiques, ont progressivement migré vers des réseaux dits numériques. La numérisation de l'information consiste à coder l'information sous la forme d'un flux binaire, c'est-à-dire une succession de 0 et 1. La communication s'établit entre deux parties situées à des extrémités du réseau. Les deux parties partagent un algorithme qui code l'information sous une forme numérique. Le récepteur peut donc décoder le flux binaire qu'il reçoit et reconstituer l'information d'origine. L'émetteur et le récepteur peuvent alors échanger et partager de l'information.

Les réseaux de télécommunications sont constitués d'équipements électroniques permettant le transport du flux binaire sur différents média tels que des ondes radios, des fils de cuivre ou de la fibre optique. Afin que le flux binaire puisse être transporté de façon optimale, en un minimum de temps et sans dégradation, il est souvent compressé puis codé pour le médium utilisé.

Parallèlement aux réseaux de télécommunications, la démocratisation de l'ordinateur a permis l'avènement des réseaux informatiques. L'agrégation des réseaux informatiques forme le réseau Internet. Internet est ainsi un réseau de réseaux informatiques à l'échelle mondiale. Ce réseau est très similaire dans sa structure aux réseaux de télécommunications et repose sur les mêmes média de communications. La différence principale entre ces réseaux se situe dans les protocoles mis en œuvre, c'est-à-dire l'ensemble des règles et des messages à respecter lors des communications, et la capacité de contrôle des équipements au sein du réseau. Les réseaux de télécommunications sont standardisés par l'UIT, Union International des Télécommunications. Les réseaux informatiques sont eux standardisés par l'IETF, Internet Engineering Task Force. Il en résulte deux types de réseaux très hétérogènes ne permettant pas l'interopérabilité. De plus, il y a une différence fondamentale de conception entre ces deux types de réseaux. D'un côté, les équipements des réseaux de télécommunications sont complexes et contrôlent des terminaux basiques tels que des téléphones. De l'autre côté, les équipements des réseaux informatiques effectuent des tâches simples, se contentant essentiellement d'acheminer l'information, tandis que les terminaux, comme des ordinateurs par exemple, sont complexes et contrôlent les communications.

Ces dernières décennies sont apparus les téléphones mobiles, les assistants numériques personnels (PDA pour Personal Digital Assistant) et la téléphonie via Internet. À chaque nouvelle génération, de nouvelles fonctionnalités sont ajoutées à ces périphériques. D'un côté les terminaux de télécommunications se complexifient et permettent désormais l'envoi de messages

textuels, la navigation sur le *Web*, l'utilisation du courrier électronique. De l'autre côté, des protocoles informatiques permettent de remplir la fonction historique des réseaux de télécommunications grâce à la téléphonie via Internet. On observe une convergence des fonctionnalités de ces deux grands réseaux avec une migration progressive vers un réseau unique et informatique.

L'utilisation, depuis de nombreuses années, de ces réseaux a profondément bouleversé les habitudes et les communications entre êtres humains. Les communications jouent désormais un rôle central dans les sociétés que ce soit pour des communications personnelles d'ordre privé ou professionnel, pour des communications lors d'échanges économiques (bancaires ou boursiers par exemple), ou encore pour des communications militaires (entre des troupes et leur état major par exemple). Ce nouveau rôle des communications tend à multiplier les échanges de données et rendre omniprésentes les communications. Ainsi, nous sommes de plus en plus sollicités au téléphone, par courrier électronique, par messagerie instantanée, par messages courts (SMS). Une assistance devient nécessaire pour faire face à la multiplication des moyens de communications et à l'augmentation de leur usage. Cette assistance doit faciliter la gestion des communications en automatisant certains traitements de l'information et ainsi rendre un service à l'utilisateur. Par exemple, un service de filtrage de courriers électroniques permet de rejeter automatiquement les courriers indésirables.

Nous nous attacherons dans ce chapitre à caractériser les communications ainsi que les services de communications associés. L'objectif est de pouvoir les catégoriser afin que leur hétérogénéité ne soit plus un frein à leur compréhension.

2.1 Caractérisation des communications

Les communications sont caractérisées par plusieurs critères : la variété des types de données échangées et les différentes façons de réaliser les échanges ; nous parlerons de modes de communications. Enfin, les communications se définissent en fonction des entités communicantes mises en jeux lors de la communication.

2.1.1 Types de données

Le but d'une communication est d'échanger de l'information entre l'émetteur et le récepteur. L'information échangée peut être un texte, de la voix, une vidéo mais aussi une mesure numérique, envoyée par exemple par un capteur de température. Lorsque l'information est un texte, celui-ci peut prendre plusieurs formats, plus ou moins structurés allant du texte brut à du LaTeX [MGB+04] ou de l'ODF (Open Document Format) [OAS07] en passant par du HTML (HyperText Markup Language) [W3C02] ou du XML (eXtensible Markup Language) [BPSM+06]. De la même manière, il existe plusieurs formats pour les autres types de données comme la vidéo, le son ou les images.

Les flux multimédia, souvent audio et vidéo, sont particulièrement volumineux. Afin de réduire l'espace disque ou le temps de transfert, il est courant de les compresser. Il existe pour cela plusieurs dizaines de méthodes. Chaque méthode de compression est bien évidemment réversible afin de pouvoir restituer l'information d'origine, éventuellement dégradée. On parle d'encodeur pour le logiciel réalisant la compression et de décodeur lors de la décompression. Un codec, contraction des termes codeur et décodeur, fait référence à l'ensemble des deux logiciels. Pour que deux interlocuteurs soient capables de communiquer, ils doivent préalablement convenir du format des données à échanger et donc du codec à utiliser.

Dans les réseaux téléphoniques historiques, il n'existe que deux normes, l'encodage utilisant la loi A, G.711 A-law, et celui utilisant la loi μ, G.711 μ-law. La première est utilisée en Europe tandis que la seconde est utilisée en Amérique du nord et au Japon. L'interopérabilité des réseaux permettant à un français de téléphoner à un américain ou à un japonais n'est donc pas compliquée puisqu'il n'y a que deux formats utilisés. Dans la téléphonie via Internet, il existe une multitude de formats utilisables tels que Speex [Spea], iLBC [ADA+04], G.729, GSM ou encore le format G.711 sous ses deux variantes. Il devient alors beaucoup moins facile de garantir l'interopérabilité du système. Certains codec, comme le G.729, sont protégés par des brevets et leur utilisation requiert le versement de royalties. D'autres n'ont pas leur spécification publique. De fait, certains codec ne sont pas ou ne peuvent pas être présents sur l'ensemble des téléphones utilisés pour la téléphonie via Internet. Il devient dans ces conditions plus difficile de pouvoir garantir les communications entre des intervenants.

On retrouve le problème d'interopérabilité pour d'autres types de données. Le mécanisme de codec est en effet également utilisé pour compresser les images, qu'il s'agisse d'image unique ou de séquence d'images, c'est-à-dire de vidéo. Les images peuvent être stockées, par exemple, sous des formats tels que BitMaP (BMP), JPEG File Interchange Format (JFIF), Graphics Interchange Format (GIF), Portable Network Graphics (PNG), Scalable Vector Graphics (SVG), Windows Meta File (WMF), Tagged Image File Format (TIFF). Les vidéos quant à elles sont encodées avec des formats tels que Cinepak, H.263, MPEG-2, MPEG-4, RealVideo, Theora, Windows Media Video. Ces listes ne sont *absolument* pas exhaustives et évoluent avec l'apparition de nouveaux codec.

Chaque type de représentation de l'information présente des avantages et des inconvénients. Tous ces formats sont donc utilisés, même si en pratique quelques uns dominent en terme d'usage. Il est vital dans une société fondée sur les communications et les échanges de garantir le plus tôt possible la compréhension par le récepteur de l'information émise et ce malgré l'abondance des types de données et de leur format. Garantir le succès d'une communication permet d'éviter les échanges inutiles. La congestion du réseau et les traitements inutiles de la part des intervenants sont ainsi éludés.

2.1.2 Modes de communications

L'échange de données lors d'une communication peut s'effectuer de plusieurs manières. Les données peuvent être envoyées spontanément par l'initiateur de la communication, ou au contraire, réclamées par celui-ci. Dans les deux cas, il peut poursuivre d'autres traitements en attendant une réponse, ou au contraire, attendre une réponse afin de pouvoir poursuivre son propre traitement. Il y a donc deux plans à distinguer, chacun ayant un rôle particulier : le plan de contrôle, qui gère la communication, et le plan de données, qui achemine de l'information. Nous allons à présent voir les différentes possibilités qui existent pour chacun de ces deux plans.

2.1.2.1 Plan de contrôle

Le plan de contrôle gère le cycle de vie des communications plutôt que leur contenu, c.-à-d. les données qui circulent. Le plan de contrôle permet d'identifier : l'initiateur d'une communication, les principes régissant l'échange de données, et si l'opération de communication est bloquante ou pas. Il existe deux types de contrôle, les communications synchrones et les communications asynchrones.

Communications synchrones Lorsqu'un responsable demande un renseignement à un employé, il contrôle la communication mais l'information est donnée par l'employé. Dans le cas d'une information simple et connue de l'employé, le responsable est en attente d'une réponse immédiate. L'employé doit alors fournir l'information demandée avant de pouvoir reprendre sa tâche précédente.

De manière similaire, lors d'un appel téléphonique, c'est l'appelant qui initie la communication. Les deux interlocuteurs peuvent ensuite converser et, dans la plupart des cas, ne font rien d'autre en même temps. À la fin de la conversation, les interlocuteurs raccrochent. Chacun peut alors reprendre une autre activité.

Lorsque, comme dans ces deux exemples, un intervenant, ou l'ensemble des intervenants, est monopolisé par la communication, on parle de communications synchrones, ou bloquantes.

Communications asynchrones Lorsqu'un responsable souhaite diffuser une note de service, il a l'initiative de la communication et ne s'assure pas que la note est bien lue par les employés. De même, lors de l'envoi d'un courrier électronique, l'expéditeur n'attend pas de retour pour savoir si le courrier électronique est bien arrivé. Dans ces deux cas, l'initiateur de la communication ne reste pas en attente d'une réponse. On parle dans ce cas de communications asynchrones, ou non bloquantes.

Nous allons à présent voir un autre exemple de communication asynchrone. La différence réside dans le fait que l'information n'est pas émise par l'initiateur de la communication mais au contraire, l'information est réclamée par celui-ci. Lorsqu'un responsable demande un rapport à un employé, il contrôle la communication mais l'information est donnée par l'employé. Le rapport doit préalablement être réalisé ce qui peut prendre un certain temps. Le responsable ne s'attend donc pas à recevoir immédiatement le rapport. Lorsque l'employé est en mesure de fournir le rapport demandé, il le transmet à son responsable.

Dans une situation telle que celle décrite ci-dessus, on parle également de communications asynchrones. En effet, le responsable n'attend pas oisivement le rapport mais peut s'affairer à d'autres tâches. L'employé le notifiera plus tard, lorsque le rapport sera prêt. Le fait que l'employé notifie son responsable, alors que c'est ce dernier qui a initié la communication, constitue une inversion du contrôle. C'est en effet l'employé qui acquiert, dans une certaine mesure, le contrôle de la communication et du moment où il fournira le rapport.

Les communications synchrones caractérisent le fait que l'intervenant qui initie la communication reste en attente de la réponse de son interlocuteur. Tandis que les communications asynchrones caractérisent le fait que l'intervenant qui initie la communication, peut exécuter d'autres tâches immédiatement. Lorsque l'initiateur de la communication souhaite recevoir une information, il demande à être notifié a posteriori.

2.1.2.2 Plan de données

Nous allons à présent analyser comment circulent les données entre les intervenants en fonction des différents types de contrôle identifiés précédemment.

Communications synchrones Lorsque le responsable demande une information à l'employé, les données circulent de l'employé vers le responsable. Toutefois, il est fréquent que des données proviennent du responsable. Ce dernier fournit un contexte à l'employé. Si le responsable demande le nombre de participants à une réunion, il est probable qu'il donne également

une information de contexte comme une date et éventuellement un horaire. De ce simple exemple, nous pouvons déduire que les données peuvent circuler dans les deux directions entre les intervenants.

Lors d'un appel téléphonique, les données circulent également de façon bidirectionnelle. Mais contrairement à l'exemple précédent où les échanges se suivaient en séquence, l'échange d'information forme deux flux de données audio, c'est-à-dire une succession d'échantillons de voix. Ces deux flux sont simultanés et de sens opposé. Les deux interlocuteurs peuvent parler en même temps. Même si, dans le cas d'interlocuteurs humains, cela présente peu d'intérêt, dans le cas de communications entre deux équipements électroniques, cette capacité à communiquer simultanément dans les deux sens permet d'optimiser la communication. Par exemple, lors de la synchronisation entre deux systèmes, chacun peut envoyer les nouvelles informations pendant qu'il reçoit celles de l'autre.

Nous retiendrons de ces exemples trois principes. Le plan de données est orienté entre un émetteur et un récepteur. L'initiateur de la communication peut fournir un contexte s'il n'est pas l'émetteur. Et, lors de communications de flux de données, les deux intervenants peuvent être à la fois émetteur et récepteur.

Communications asynchrones Nous avons précédemment vu trois exemples de communications asynchrones, la diffusion d'une note de service, l'envoi d'un courrier électronique et la demande d'un rapport. Dans les deux premiers exemples, les données sont émises par l'initiateur de la communication. Au contraire, dans le troisième exemple, les données, c'est-à-dire le rapport, ne sont pas émises par l'initiateur mais par son interlocuteur, l'employé.

Le plan de données est orienté de l'émetteur vers le récepteur, comme c'est le cas avec une communication synchrone. De plus, l'initiateur peut être également l'émetteur des données. Les données peuvent donc être échangées dans les deux sens vis à vis de l'initiateur de la communication lors de communications asynchrones. Nous retiendrons également qu'il n'existe pas de communication asynchrone pour l'échange de flux de données car l'échange d'un flux de données nécessite une synchronisation entre les interlocuteurs afin qu'ils négocient les paramètres du flux.

Nous pouvons à présent classer les communications en deux grandes familles : d'une part les communications synchrones et d'autre part les communications asynchrones. Afin de bien identifier le mode de communications utilisé lors d'un échange de données, nous raffinerons la description en orientant le plan de données.

2.1.3 Entités communicantes

Dans la section 2.1.1, nous avons vu les différents types de données échangées et dans la section précédente, comment ces données sont échangées. Nous allons à présent caractériser les intervenants. Ces intervenants ont vocation par nature à communiquer, nous utiliserons également à partir de maintenant le terme d'entités communicantes pour les désigner.

Il existe deux types d'entités communicantes, les êtres humains et les machines. La modélisation des communications humaines sortant du cadre de cette étude, nous les prendrons en compte grâce à la modélisation des dispositifs qui interagissent avec des êtres humains. Nous modélisons indirectement les communications humaines en modélisant des dispositifs comme les téléphones, les écrans, les caméras et les imprimantes.

Caractériser les entités communicantes revient alors à caractériser les dispositifs matériels et logiciels qui communiquent. Pour caractériser les entités, il faut être capable de les décrire en terme de propriétés et de moyens de communications. Nous utiliserons le terme de *fonctionnalité* pour faire référence aux divers moyens de communications.

2.1.3.1 Propriétés des entités

Tous les dispositifs ont des caractéristiques, des spécifications qui les différencient les uns des autres. Tous les ordinateurs n'ont pas forcément la même capacité de mémoire vive. La résolution, les codec ou la profondeur d'image d'une caméra sont autant d'éléments qui la caractérisent.

Ces propriétés sont des données décrivant les entités. Il est possible d'utiliser les types d'informations présentés dans la section 2.1.1 pour représenter les données.

Les propriétés représentent des caractéristiques statiques des entités comme la vitesse du processeur pour un ordinateur ou les codec disponibles sur une caméra. Toutefois certaines sont dynamiques. Un PDA par exemple est un périphérique mobile, de même qu'un téléphone GSM ou WiFi (Wireless Fidelity). Il est intéressant de pouvoir localiser ces objets. Il est ainsi possible de localiser indirectement leur propriétaire ou d'indiquer à celui-ci où se trouve son bien. La propriété indiquant la localisation peut donc être dynamique pour certaines entités.

Nous retiendrons que les entités communicantes sont caractérisées à l'aide de propriétés. Les propriétés sont des données dont la représentation est déterminée par un type. De même que l'entité évolue dans le temps, certaines de ses propriétés évoluent. Il y a donc des propriétés statiques et d'autres dynamiques.

2.1.3.2 Fonctionnalités

Nous avons vu comment définir les caractéristiques d'une entité à l'aide de propriétés. Il reste cependant des caractéristiques primordiales à définir pour chaque entité. Il s'agit de ses moyens de communications. Comment communique une entité ? Qu'échange-t-elle comme données ?

Une fonctionnalité est l'association d'un mode de communications et d'un type de données. Elle décrit ainsi comment un certain type de données est échangé. Une entité dispose de plusieurs fonctionnalités. Il est ainsi possible d'échanger différentes données de différentes façons. Un ordinateur peut ainsi être vu à la fois comme une entité d'affichage pour des images ou des vidéos, une entité de capture audio, une entité de rendu audio et une entité de stockage. Le corollaire est qu'il est possible d'échanger la même donnée de différentes façons. Un capteur de température peut ainsi être consulté à la demande, de manière synchrone, ou émettre de lui-même des mesures de température à des entités qui en auraient fait la demande. Il s'agirait alors d'une communication asynchrone. Enfin, un capteur peut communiquer un flux de mesures à une entité, pour réaliser un asservissement par exemple.

Nous pouvons définir les communications selon trois critères : (i) le type des données échangées, (ii) le mode de communications utilisé et (iii) les entités qui communiquent. Les entités communicantes sont de type très différentes : téléphones, capteurs de mesure, caméras, écrans, équipements électroménagers. Il y a donc un problème pour les modéliser et pouvoir suivre leur évolution. L'apparition constante de nouvelles entités est un facteur aggravant. Cette pléthore d'entités communicantes induit une augmentation des communications. Nous

pouvons caricaturer en disant que tous les objets communiquent tout le temps. Nous nous approchons chaque jour un peu plus de cette caricature.

2.2 Caractérisation des services de communications

Pour faire face à l'explosion du nombre de communications et de sollicitations, l'automatisation de certaines tâches devient impérative. Une solution pour automatiser des traitements consiste à développer des services de communications. Les services de communications se distinguent en deux classes de services. D'un côté, la classe des services de routage regroupe les services qui modifient la communication pendant son transit dans le réseau. D'un autre côté, la classe des services entités regroupe les services qui ont le rôle d'un intervenant dans la communication. Les services entités sont ainsi capables de réagir et traiter les stimuli de communications. Ils peuvent de plus générer des communications vers d'autres services.

2.2.1 Services de routage

La caractéristique principale de la classe des services de routage est que l'initiateur de la communication n'a pas connaissance de leur présence. Il ne peut d'ailleurs pas, dans la plupart des cas, la soupçonner. La notion de service de routage est transversal par rapport aux types de données. On trouve ainsi des services de routage dans des domaines aussi variés que le courrier électronique et la téléphonie.

Courrier électronique Qui voudrait encore filtrer son courrier électronique indésirable, ou *spam*, manuellement ? Il est déjà de plus en plus difficile de faire face aux courriers traditionnels. Pour lutter contre le problème du courrier indésirable, il existe des solutions permettant de trier, et éventuellement supprimer ce courrier automatiquement [Spa]. Ces services *anti-spam* sont souvent réalisés au niveau du serveur de courrier électronique dans le réseau pour l'ensemble des usagers d'une organisation. Ce type de services agit sur une communication initialement prévue entre l'expéditeur du courrier et l'un des destinataires. Le service de filtrage du courrier ne fait que modifier la communication en déplaçant le spam en dehors du courrier entrant, c'est-à-dire dans une autre boîte aux lettres. Cette opération qui consiste à grouper et trier le courrier en fonction d'un critère, s'appelle le *routage*. On utilise par extension le terme routage pour toutes les opérations qui consistent à déterminer la destination d'une information. Il existe ainsi le routage de paquets IP ou encore le routage d'appels.

Un autre exemple de service de routage du courrier concerne l'utilisation des listes de diffusion. Il est fréquent d'être membre d'une liste de diffusion et de recevoir ainsi des courriers électroniques destinés, par exemple, à l'ensemble d'un service administratif dans une organisation. Il est alors pratique de séparer ce type de courrier, du courrier plus personnel, en le plaçant directement dans une autre boîte aux lettres que celle du courrier entrant. Il existe, pour faciliter et permettre aux utilisateurs de personnaliser le routage de leurs courriers, des solutions comme Sieve [GS08]. Il s'agit d'un petit langage à script permettant de router le courrier en fonction des propriétés de celui-ci. Il est ainsi possible de router tous les messages provenant d'une personne, ou ayant une thématique commune, dans une boîte aux lettres particulière.

Téléphonie Il existe le même genre de service de routage pour les appels téléphoniques. Il est ainsi possible de systématiquement refuser les appels anonymes ou de rediriger les appels en

cas d'absence ou de non réponse vers un autre numéro. Enfin, dans les entreprises disposant de leur propre réseau téléphonique pour les appels internes, il est possible de prendre les appels d'un téléphone qui sonne, à partir d'un autre téléphone ou de faire sonner l'ensemble des téléphones d'un bureau (bien que chacun d'eux ait son propre numéro).

2.2.2 Services entités

La classe des services entités identifie les services qui se comportent comme une entité communicante. Ils peuvent envoyer un courrier électronique ou répondre au téléphone comme c'est le cas avec un répondeur téléphonique. Enfin, dans les systèmes ubiquitaires, toutes les entités sont communicantes et capables d'interagir avec leur environnement.

Courrier électronique Il existe des services de courrier électronique qui agissent comme une entité. Lorsqu'une personne part en vacances, elle peut configurer sa boîte aux lettres pour qu'un courrier, indiquant par exemple sa date de retour, soit automatiquement envoyé aux personnes qui lui écrivent. Le service gérant l'envoi du courrier automatique en cas d'absence se comporte comme une entité communicante.

Il existe d'autres services utilisant le courrier électronique comme moyen de communications. Par exemple, il existe des logiciels pour la gestion de liste de diffusion tel que Mailman [Mai] ou Majordomo [Maj]. La particularité de ces gestionnaires est qu'ils autorisent la gestion des listes de diffusion par une adresse de courrier électronique particulière. Il est ainsi possible d'administrer la liste `maliste@mondomaine.com` via des courriers envoyés à l'adresse `maliste-request@mondomaine.com`. Le logiciel de gestion des listes est alors une entité à double titre. D'une part, il répond aux courriers sur l'adresse des requêtes, et d'autre part, il copie les courriers adressés à la liste à chacun des membres.

Téléphonie De manière similaire, il existe des services de téléphonie qui se comportent comme des entités communicantes. C'est le cas notamment avec, par exemple, un répondeur vocal. Il décroche automatiquement, éventuellement après un délai d'attente, puis fait une annonce. Enfin, il enregistre le message de l'appelant et raccroche. Il a fallu attendre près de soixante ans après l'invention du téléphone pour voir apparaître le premier répondeur en 1935. Près de soixante ans après, de nouveaux services apparaissent encore. Depuis quelques années, il est fréquent de trouver un menu vocal lorsque l'on téléphone au service client d'une entreprise ou à un organisme publique. Plus récemment, ces menus vocaux se sont enrichis de synthèse vocale. Il est désormais possible d'écouter son courrier électronique grâce à un service téléphonique.

Systèmes informatiques ubiquitaires Les systèmes informatiques ubiquitaires sont des systèmes composés d'une multitude de périphériques électroniques. Ces périphériques sont dispersés dans les environnements qui nous entourent, la maison, le bureau, l'atelier d'usine, la rue, la voiture, etc. Ils ont la particularité de pouvoir communiquer. Ils utilisent pour cela différents média, réseaux filaires et non-filaires. Ces périphériques sont donc des entités communicantes au sens où nous l'avons défini. Ces entités peuvent aller d'une simple lampe à l'électroménager, en passant par les équipements multimédia.

Une pléthore de services est alors possible comme surveiller la cuisson du rôti depuis sa télévision pendant que l'on regarde un film ou mettre automatiquement sur pause un film avant même que le téléphone ne sonne et prendre l'appel sur son téléviseur. Le réfrigérateur peut

diffuser des messages d'alerte sur un écran indiquant les produits périmés ou en passe de l'être. Il peut également faire l'inventaire des produits à ne pas oublier lors des prochaines courses voire même passer commande automatiquement sur un site Internet marchand via l'ordinateur. Tous ces scénarios encore futuristes seront très prochainement possibles. Il convient de s'y préparer au mieux et d'appréhender le plus tôt possible les problèmes qui vont survenir.

L'ensemble des entités présentées communiquent des informations de nature très différente. Le développement de services permettant la coordination de ces entités est un vrai défi. En effet, la programmation de tels services nécessite de pouvoir manipuler des types de données arbitraires. De plus, les données sont échangées selon les différents modes de communications que nous avons vus. Enfin, les entités disposent de fonctionnalités spécifiques pour communiquer qui dépendent de leur nature intrinsèque. Actuellement, le développement de services ubiquitaires est ad hoc. L'approche actuelle ne permet pas la réutilisation des services déjà écrits, ni la coopération de ces derniers pour fournir de nouveaux services plus riches.

Il est important de bien identifier et différencier ces deux types de services : les services de routage et les services entités. En effet, les services de routage opèrent lors de la communication entre des entités et utilisent pour prendre leur décision des caractéristiques de la communication tels que l'expéditeur, le destinataire, le sujet, l'heure. Généralement, ils n'utilisent pas et ne savent pas manipuler les données qui transitent lors de la communication. Les services entités sont des entités communicantes et sont donc à ce titre en mesure d'utiliser, de manipuler ou d'émettre des données pendant la communication.

2.3 Bilan

Les communications prennent une place prépondérante dans notre société. Tandis que les communications se multiplient et se complexifient, il faut pouvoir les maîtriser. Il est donc nécessaire d'automatiser certains traitements. Le traitement automatique des communications constitue un service. Le développement de tels services pose plusieurs problèmes. Il faut pouvoir tenir compte de la grande hétérogénéité des communications tant au niveau des entités communicantes que de leurs fonctionnalités, c'est-à-dire des types de données et des modes de communications. De plus, les communications que nous considérons s'effectuent par nature entre des entités distantes. Elles forment un système distribué, construit autour d'un réseau. La programmation de services de communications doit faciliter la coordination entre les services présents dans le réseau. Le modèle de programmation doit abstraire les problèmes liés à la distribution des données ainsi qu'à l'hétérogénéité des communications. Enfin, en fonction de l'objectif d'un service, un développeur doit mettre en œuvre, soit un service de routage des communications, soit un service entité qui se comporte comme une entité communiquante.

Les réseaux informatiques fournissent un service de transport de données. Ce transport des données constitue le support de base des communications. Pour permettre des communications particulières sur un tel réseau, des protocoles applicatifs ont été définis. Ils normalisent les échanges en fonction de l'objectif d'une application donnée. La normalisation des protocoles applicatifs fournit aux utilisateurs un minimum d'interopérabilité entre les systèmes. Dans le chapitre suivant, nous allons présenter quelques uns de ces protocoles réseaux permettant de gérer des communications au niveau applicatif.

Chapitre 3

Services de téléphonie IP

Le domaine de la téléphonie a vécu une révolution avec la convergence des réseaux de télécommunications et des réseaux informatiques. Les fonctionnalités de la téléphonie IP dépassent désormais un simple appel téléphonique. La téléphonie IP intègre des fonctionnalités de présence et de messagerie instantanée ainsi que le support de la vidéo. De fait, la téléphonie IP illustre parfaitement la diversité des types de communications que nous venons de voir. De plus, à l'instar de la téléphonie traditionnelle, il s'agit d'un domaine où de nombreux services sont possibles. Le développement de ces services pose cependant plusieurs problèmes. Les services doivent-ils être déployés dans le réseau ou sur les terminaux ? L'ajout de programmes utilisateur dans le système de téléphonie est une source d'erreur. Comment préserver le comportement nominal d'un système de téléphonie en présence de services ? Tout en présentant différents systèmes de téléphonie IP, nous chercherons celui capable d'offrir l'environnement le plus propice au développement de services.

3.1 Téléphonie IP

Les systèmes de téléphonie IP sont fondés sur les réseaux informatiques. Après une courte introduction aux réseaux informatiques et à la téléphonie, nous verrons divers protocoles de téléphonie IP. À l'instar des protocoles en général, ils se répartissent en deux grandes catégories, d'une part les protocoles binaires et d'autre part les protocoles textuels. Nous terminerons cette section par la présentation d'un protocole particulier de téléphonie, le protocole SIP *(Session Initiation Protocol)*.

3.1.1 Introduction aux réseaux informatiques

Dès les premiers systèmes informatiques, une des premières préoccupations fut de les interconnecter. Des ensembles de règles et de conventions ont donc été formalisés pour définir les messages échangés, et notamment leur format. Ces ensembles de règles et de conventions forment des protocoles. Chaque protocole remplit une fonction particulière qui sert de fondation à un autre protocole. On obtient ainsi un empilement protocolaire. Le modèle TCP/IP, voir Figure 3.1, est l'empilement protocolaire de référence pour Internet. Il est fondé sur les protocoles du même nom, TCP [Pos81b] *(Transmission Control Protocol)* et IP [Pos81a] *(Internet Protocol)*. La première couche, la couche d'accès au réseau physique, se décompose en deux fonctionnalités. D'une part, elle définit le protocole codant le signal sur le médium, par exem-

ple des fils de cuivre ou une fibre optique. D'autre part, elle assure la gestion de la connexion entre deux équipements directement reliés par ce médium, par exemple un ordinateur et un commutateur. Le protocole Ethernet [Ass05] est un exemple connu de protocole de la couche d'accès au réseau. Le protocole IP est un protocole de la seconde couche, la couche réseau. Il assure la connexion entre des domaines, par exemple `inria.fr` et `labri.fr`. Les machines sont identifiées par une adresse IP. Enfin, le protocole TCP est un protocole de la couche transport. Il fournit une connexion de bout en bout, entre deux systèmes informatiques d'un réseau informatique, ou réseau IP. Une connexion est caractérisée localement par un port de communication. Cette connexion de bout en bout est ainsi définie par quatre éléments : deux adresses IP et deux ports de communication. Le réseau Internet est fondé sur l'empilement de la couche transport TCP au-dessus de la couche réseau IP. Au-dessus de la couche transport se trouve la couche application. Le rôle de cette couche est de fournir un service spécifique tel que le courrier électronique, le Web ou la téléphonie. Afin de rendre des services spécifiques, un protocole applicatif, c'est-à-dire un protocole de la couche application, est défini pour chaque service. On trouve ainsi les protocoles SMTP, POP et IMAP [Pos82, MR96, Cri96] pour le courrier électronique. Chacun remplissant une tâche particulière dans l'acheminement du courrier entre l'expéditeur et le destinataire. Le service du Web est réalisé grâce au protocole HTTP [FGM+97]. Enfin, depuis la convergence des réseaux de télécommunications et des réseaux informatiques, la téléphonie est assurée par des protocoles applicatifs.

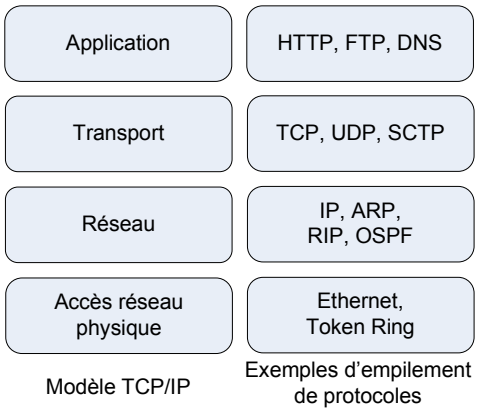


FIG. 3.1: Le modèle TCP/IP

3.1.2 Introduction à la téléphonie

Les réseaux de télécommunications offrent historiquement le service téléphonique grâce à des protocoles patrimoniaux tels que le Réseau Téléphonique Commuté (RTC) ou le Réseau Numérique à Intégration de Services (RNIS). Le développement de ces technologies a permis de caractériser un appel téléphonique. Un appel téléphonique est constitué de deux plans : un plan de contrôle qui gère l'appel, c'est-à-dire son établissement et sa terminaison, et un plan usager qui transporte les données d'un interlocuteur à l'autre. Les opérations du plan de contrôle forment la *signalisation*. La signalisation permet de construire un circuit virtuel entre l'appelant et l'appelé. Ce circuit virtuel véhicule ensuite les données de l'appel. Au début de la téléphonie, les deux plans n'étaient pas disjoint et la signalisation était *mélangée* aux

données audio. On parle dans ce cas de signalisation en bande. Pour des raisons pratiques, ces deux plans sont désormais disjoints et la signalisation est décorrélée des données de l'appel. On parle alors de signalisation hors bande. La convergence des réseaux de télécommunications et des réseaux informatiques a été l'occasion de voir apparaître des protocoles de signalisation fonctionnant sur les réseaux IP.

3.1.3 Protocoles binaires de téléphonie IP

Les protocoles binaires sont plus compacts et proche de l'ordinateur. Les informations et les opérations sont représentées sous une forme binaire, c'est-à-dire une succession de 0 et de 1 facilement manipulable par une machine. Les protocoles binaires sont les premiers à être apparus. Ils répondent aux fortes contraintes historiques en terme de puissance de calcul et d'espace mémoire disponible. Bien que les processeurs modernes permettent de s'en affranchir, ils restent pertinents dans des contextes de hautes performances, par exemple lorsque le nombre d'appels est important. Les principaux protocoles de téléphonie binaires sont H.323 [MMS00], Skype [Sky] et IAX [SCG+08].

3.1.3.1 H.323

Le protocole H.323 est un standard de l'UIT-T, le groupe de travail dédié à la téléphonie au sein de l'Union Internationale des Télécommunications. Il regroupe un ensemble de protocoles permettant le déploiement d'un système téléphonique au-dessus d'un réseau IP. Chacun des protocoles qu'il regroupe, a une tâche spécifique allant du contrôle de la communication à l'encodage de la voix, ou de la vidéo en passant par l'échange d'autres types de données. L'utilisation de certains de ces protocoles sont optionnels, par exemple ceux traitant de la qualité de service, tandis que d'autres sont obligatoires afin d'avoir un système fonctionnel.

Le protocole H.323 est dédié à la téléphonie et n'offre pas de possibilité simple d'extensions en vue de supporter une grande variété de services de communications. De plus, sa mise en œuvre est complexe. L'utilisation d'un protocole binaire nécessite en effet de maintenir des outils spécifiques pour le développement et la mise au point du protocole et de ses extensions. Maintenir de tels outils est aussi complexe que de développer les clients et les serveurs définis par le protocole.

3.1.3.2 Skype

Skype est le nom d'un logiciel édité par la société Skype Technologies. Ce logiciel a donné son nom au protocole applicatif qu'il implémente. Le protocole Skype est un protocole pair à pair. C'est-à-dire qu'il fonctionne de manière fortement distribuée : la présence de serveurs administrés par la société est minime et la majeure partie du réseau fonctionne grâce aux utilisateurs eux-mêmes. En effet, le simple fait d'exécuter le logiciel, afin de pouvoir être joint, transforme l'ordinateur de l'utilisateur en plate-forme téléphonique capable de router des appels et de relayer des communications audio et vidéo.

Le protocole Skype est un protocole fermé, il n'existe aucune spécification publique. De plus, l'unique implémentation disponible est protégée par des techniques permettant d'obscurcir le code binaire du programme et ainsi limiter la rétro-analyse du protocole. De fait, il est actuellement impossible à des parties tierces non accréditées d'utiliser le réseau Skype. L'ajout de services par l'utilisateur ou un tiers quelconque n'est donc pas possible dans ce contexte.

3.1.3.3 Inter-Asterisk eXchange

Asterisk [Speb] est un commutateur téléphonique logiciel. Il est capable de gérer à la fois des protocoles de téléphonie patrimoniaux et des protocoles de téléphonie IP. Cette double compétence lui confère également une fonctionnalité de passerelle entre différents réseaux de téléphonie.

En plus de protocoles existants, Asterisk propose son propre protocole de téléphonie, IAX. Il a été conçu afin de fonctionner simplement, y compris en présence de traduction d'adresses réseaux (en anglais *Network Address Translation* ou NAT), et d'utiliser au mieux la bande passante du réseau. C'est pour ces raisons qu'il s'agit d'un protocole binaire fonctionnant sur un unique port utilisé à la fois pour la signalisation et le transport du flux multimédia. Ce choix de conception limite l'extensibilité. Le nombre de codec définis est limité par la structure même du protocole. De plus, les codec utilisables pour une communication sont ceux disponibles sur l'ensemble des nœuds traversés par la signalisation d'un appel. Enfin, l'utilisation d'un port unique, d'un protocole unique pour la signalisation et le transport des média, ainsi que la présence d'un serveur dans certains cas pendant les communications pose un problème face aux attaques de déni de service et limite le passage à l'échelle.

IAX est un protocole binaire de téléphonie supportant la voix et la vidéo. Il est cependant difficilement extensible et n'est pas encore un standard. Il n'est donc pas adapté pour le développement de services de communications et le transport d'un type arbitraire de données.

3.1.4 Protocoles textuels de téléphonie IP

Les protocoles textuels sont plus proches d'une représentation humaine. Les informations et les opérations sont représentées par du texte clair et lisible par un humain. Ainsi, les protocoles textuels procurent une grande souplesse lors de leur mise au point. De plus, ils sont par nature beaucoup plus facilement extensibles que les protocoles binaires. En contrepartie, ils nécessitent plus de puissance de calcul et d'espace mémoire. Les protocoles textuels MSNP [MSN06], XMPP [SA04] et SIP [RSC+02] offrent des fonctionnalités de téléphonie IP.

3.1.4.1 Microsoft Notification Protocol

Le protocole MSNP, MicroSoft Notification Protocol, est le protocole utilisé dans Windows Live Messenger, anciennement MSN Messenger. Il s'agit initialement d'un protocole de messagerie instantanée. Microsoft l'a fait évoluer pour ajouter des fonctionnalités de conversation audio et vidéo ainsi que du partage d'applications.

Le protocole MSNP est un protocole fermé, il n'existe pas de spécification publique officielle. Cependant, il existe plusieurs implémentations aux sources ouvertes mais aucune ne supporte l'une des trois dernières versions du protocole. L'absence de code et de spécifications à jour présente un risque de pérennité pour un projet souhaitant utiliser ce protocole. Il n'est donc pas un candidat adapté au développement des services de communications envisagés.

3.1.4.2 eXtensible Messaging and Presence Protocol

Le protocole XMPP, eXtensible Messaging and Presence Protocol, est un protocole de messagerie instantanée et de présence, anciennement nommé Jabber avant sa normalisation qui a débutée en 2002. Il fonctionne sur un modèle client/serveur mais de manière décentralisée. En effet, chaque utilisateur d'un même domaine, par exemple `inria.fr`, utilise le même

serveur, par exemple `jabber.inria.fr`. Les communications entre utilisateurs de domaines différents sont assurées par des échanges entre leur serveur respectif. Chaque serveur ne gère ainsi que les communications liées à ses utilisateurs et il n'y a pas de serveur central utilisé par l'ensemble des utilisateurs, tout domaine confondu.

Le protocole XMPP est par nature extensible et utilise une représentation XML pour ses messages. Cela pose un problème pour l'échange de données binaires. Le protocole XMPP utilise alors un encodage en base64 [Jos03] qui permet l'échange de données binaires au prix d'un message particulièrement verbeux. Il existe une extension, Jingle [XEP08], permettant de réaliser des conversations audio, fondée sur XMPP. Cependant cette extension n'est pas encore une norme et il n'existe pas d'implémentation mettant en œuvre la vidéo.

Le protocole XMPP présente de nombreux avantages pour le développement des services de communication. Il est ouvert et fait parti des standards Internet de l'IETF. Il permet l'échange de messages entre utilisateurs ainsi que des événements comme la présence des utilisateurs. Il souffre cependant d'une certaine immaturité dans le domaine des communications audio et vidéo, et plus généralement de l'échange de données binaires.

3.1.4.3 Session Initiation Protocol

Le protocole SIP, Session Initiation Protocol, est également un protocole ouvert et standard. Il est défini par l'IETF sous la RFC 3261 [RSC+02]. À l'instar de XMPP, il est extensible et de nombreuses autres RFC en étendent les prérogatives. Il s'agit initialement d'un protocole de signalisation utilisé essentiellement pour la téléphonie IP. Le protocole SIP nécessite plus de ressources matérielles qu'un protocole binaire. Cependant, il supporte assez simplement le passage à l'échelle. En effet, il peut fonctionner en pair à pair, entre deux clients, une fois la communication établie. L'établissement d'une communication peut nécessiter un ou plusieurs serveurs afin de faire aboutir l'appel. Le flux de données binaires de la conversation multimédia est ensuite transporté en temps réel par le protocole RTP [SCFJ03] *(Real-time Transport Protocol)*. Cette décorrélation, entre un protocole de signalisation et un protocole de transport en temps réel des données binaires est parfaitement adaptée au besoin des communications multimédia en général et *a fortiori* de la téléphonie IP. Les deux protocoles fonctionnent en symbiose afin de fournir à l'utilisateur le service de téléphonie IP. Ce découpage respecte la séparation d'un plan de contrôle gérant les communications et d'un plan usager transportant les données.

C'est le premier standard textuel de signalisation a avoir été normalisé par l'IETF et l'ITU. Il a ainsi été retenu dans les architectures des opérateurs de télécommunications qui opèrent en téléphonie filaire traditionnelle, cellulaire ou encore en voix sur IP. La plupart d'entre eux déploient une architecture IMS, *IP Multimedia System*, fondée sur SIP en intégrant leurs systèmes et services patrimoniaux. Cette utilisation massive par des industriels a mis en évidence les lacunes du protocole. Ils existent désormais de nombreuses extensions permettant d'améliorer la qualité de service, la sécurité et les fonctionnalités. L'extension SIMPLE [SIM] permet par exemple d'intégrer des fonctionnalités de messagerie instantanée et de présence.

Le protocole SIP est à ce jour le protocole le plus abouti et adapté pour gérer les communications dans leur diversité ; il offre, en plus de l'échange de messages sans état *à la* HTTP, des mécanismes pour les sessions et les événements. En se limitant à la signalisation, ce protocole est indépendant du type de données qui transitent dans les communications qu'il met en œuvre. Enfin, l'utilisation conjointe du protocole RTP pour les flux temps réel binaires en fait une solution parfaitement adaptée au besoin de performance lié à ce type d'applications.

3.1.5 SIP en détail

Cette étude est fondée essentiellement sur le protocole SIP. Nous allons en détailler les principes de base, puis l'architecture d'un réseau SIP et son fonctionnement. Enfin, nous aborderons quelques exemples d'extensions.

3.1.5.1 Principes de base

Le protocole SIP définit un protocole de signalisation. Nous parlerons par abus de langage de téléphonie SIP par la suite pour désigner un système de communications fondé sur le protocole SIP. Le protocole SIP réutilise les concepts introduits par les protocoles HTTP et SMTP [LCLL04]. Une représentation ASCII *(American Standard Code for Information Interchange)* est utilisée pour les messages SIP qui sont donc lisibles directement par un humain. Le protocole SIP est fondé sur le modèle de communications client/serveur décentralisé. Les messages SIP sont donc de deux types, différenciés par la première ligne, soit des *requêtes*, soit des *réponses*. La première ligne d'une requête commence par une *méthode* suivie d'une adresse Internet, URI *(Uniform Resource Identifier)* comme l'illustre la figure 3.2. Le protocole SIP réutilise également le modèle des codes de réponses HTTP tel que l'erreur *404 Not found*. Il étend les codes de réponses avec des codes dédiés à la téléphonie tel que *486 Busy here*. La figure 3.3 illustre un exemple de réponse. Une requête et ses réponses associées forment une *transaction*. Le mécanisme des en-têtes des protocoles HTTP et SMTP a également été conservé. Les en-têtes permettent de définir des couples nom/valeur permettant de caractériser l'appel. Parmi les en-têtes obligatoires dans les messages SIP, on trouve notamment les en-têtes `To` et `From`, éléments hérités du protocole SMTP. À la suite des en-têtes se trouve la charge utile, par exemple une page HTML dans le cas du protocole HTTP. La charge utile utilisée avec le protocole SIP varie en fonction de l'utilisation. Dans le cas d'un appel téléphonique, le protocole SDP *(Session Description Protocol)* est communément utilisé afin de décrire les capacités de communications des périphériques, c'est-à-dire les codec audio et vidéo supportés. Le protocole SIP se limite à la signalisation et de nouveaux protocoles, tels que SDP, sont mis en œuvre pour décrire les types de données.

L'identifiant de ligne téléphonique couramment utilisé est un numéro, le numéro de téléphone. Si l'utilisation d'un numéro est particulièrement pratique pour les commutateurs téléphoniques patrimoniaux, il n'est jamais facile d'en mémoriser soi-même une quantité importante. Enfin le développement des terminaux de téléphonie, et plus particulièrement de leur interface Homme-machine, rend l'utilisation d'un identifiant numérique obsolète. Le protocole SIP remplace les identifiants numériques par des URI. Il réutilise ainsi un mécanisme d'identification déjà largement déployé et éprouvé dans le Web et le courrier électronique. Un identifiant SIP a donc la même représentation qu'une adresse de courrier électronique, par exemple `alice@inria.fr`. Pour des raisons d'interopérabilité avec les systèmes téléphoniques existants, il est également possible d'utiliser des URI de la forme `sip:+33524574107@inria.fr`.

3.1.5.2 Architecture

Avant toute chose, les utilisateurs d'un réseau SIP doivent s'enregistrer auprès d'un serveur d'enregistrement grâce à la méthode `REGISTER`. Les utilisateurs informent via cette requête quels sont l'adresse IP et le port où ils peuvent être contactés. Si le port n'est pas explicitement

```
INVITE sip:bob@labri.fr SIP/2.0
Via: SIP/2.0/UDP alicepc.inria.fr;branch=z9hG4bK776asdhds
Max-Forwards: 70
To: Bob <sip:bob@labri.fr>
From: Alice <sip:alice@inria.fr>;tag=1928301774
Call-ID: a84b4c76e66710@alicepc.inria.fr
CSeq: 314159 INVITE
Contact: <sip:alice@alicepc.inria.fr>
Content-Type: application/sdp
Content-Length: 142

(Charge utile SDP d'Alice non représentée)
```

FIG. 3.2: Exemple de requête SIP

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP sip.labri.fr;branch=z9hG4bKnashds8
Via: SIP/2.0/UDP sip.inria.fr;branch=z9hG4bK77ef4c2312983.1
Via: SIP/2.0/UDP aliceps.inria.fr;branch=z9hG4bK776asdhds
To: Bob <sip:bob@labri.fr>;tag=a6c85cf
From: Alice <sip:alice@inria.fr>;tag=1928301774
Call-ID: a84b4c76e66710@alicepc.inria.fr
CSeq: 314159 INVITE
Contact: <sip:bob@147.210.9.15>
Content-Type: application/sdp
Content-Length: 131

(Charge utile SDP de Bob non représentée)
```

FIG. 3.3: Exemple de réponse SIP

indiqué, le port 5060 est utilisé par défaut. Par exemple, dans le cas illustré par la figure 3.4, Alice s'enregistre avec son ordinateur portable sur le serveur d'enregistrement du domaine `inria.fr`. Bob fait de même avec son téléphone Wifi et son ordinateur fixe auprès du serveur d'enregistrement du domaine `labri.fr`. Lorsqu'Alice souhaite contacter Bob, elle envoie une requête `INVITE`, telle que celle de la figure 3.2, à son serveur mandataire à l'INRIA. Celui-ci transmet la requête au serveur mandataire du domaine `labri.fr`, c'est-à-dire le serveur mandataire du destinataire. Enfin, le serveur mandataire du LaBRI transfère une copie de la requête vers chaque téléphone enregistré de Bob. Ils se mettent à sonner à la réception de la requête. Lorsque Bob décroche avec son téléphone Wifi, ce dernier envoie une réponse telle qu'illustré par la figure 3.3. La réponse passe respectivement par les serveurs mandataires du LaBRI et de l'INRIA puis arrive sur l'ordinateur d'Alice. Le protocole SIP établit une session selon la méthode dite de la *poignée de mains* en trois temps (*three-way handshake*). Après un acquittement émis par Alice (méthode `ACK`), une communication est établie entre l'ordinateur d'Alice et le téléphone Wifi de Bob. La méthode `ACK` est la seule méthode n'ayant pas de réponse associée. Cette méthode appartient à la transaction `INVITE`. À un moment arbitraire, l'un des deux raccroche mettant fin à la conversation. Il émet une requête `BYE` et la session audio s'arrête.

Afin de ce prémunir contre les pannes d'équipements réseaux, de serveurs ou de clients, le protocole SIP définit des alarmes protocolaires. Ces alarmes sont associées aux divers envois

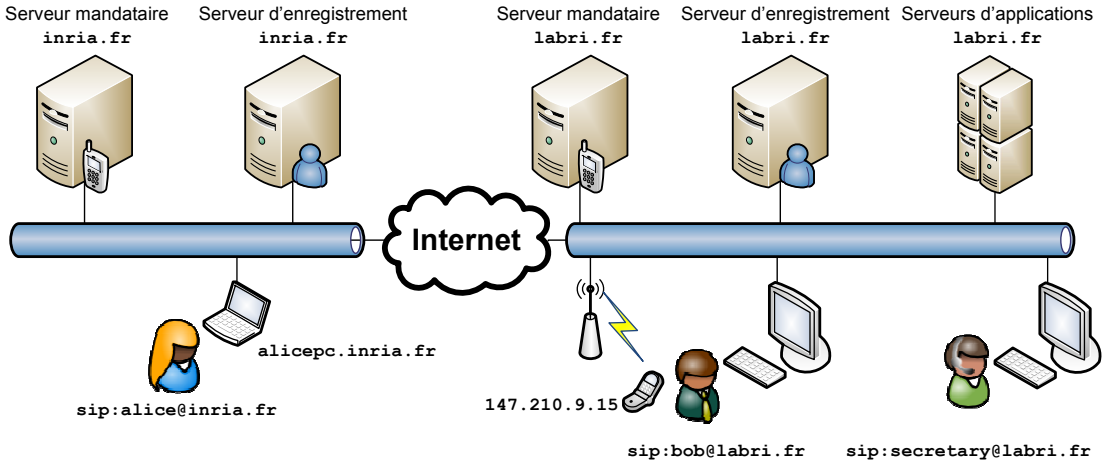


FIG. 3.4: Architecture d'un réseau SIP

de messages. Les transactions doivent se terminer avant l'expiration des alarmes. C'est-à-dire qu'une réponse doit être reçue en un temps borné. Lors de l'établissement d'une session, l'interlocuteur ne décroche pas immédiatement. Son téléphone émet des réponses provisoires qui réinitialisent les alarmes en attendant.

3.1.5.3 Extensions

Les extensions les plus connues sont celles qui ajoutent des fonctionnalités pour l'utilisateur comme la messagerie instantanée [CRS+02] (requête `MESSAGE`) ainsi que celles qui gèrent la présence [Roa02, Nie04]. La présence permet de connaître l'état de ses contacts. Il est ainsi possible de savoir si un correspondant est connecté ou non avant d'appeler. Il est nécessaire pour cela de souscrire à l'état de chacun de ses contacts grâce à la méthode `SUBSCRIBE`. À chaque changement d'état, les utilisateurs publient leur nouvel état (par exemple, libre, au téléphone ou occupé) grâce à une requête `PUBLISH`. Cette requête est envoyée au serveur de présence qui notifie alors chaque souscripteur avec une requête `NOTIFY`.

La définition de ces quatre méthodes dans le protocole SIP est générique et ne précise pas la charge utile ou la valeur de certains en-têtes. Ces méthodes fournissent ainsi des mécanismes de communications génériques et réutilisables. La messagerie instantanée et la présence, fondées sur ces mécanismes génériques, sont définies séparément. La méthode `MESSAGE` fournit une fonctionnalité similaire à celle des méthodes `GET` et `POST` du protocole HTTP. La différence principale réside dans le fait que l'échange de données à lieu entre deux interlocuteurs et non pas entre un utilisateur et un serveur. Les trois autres méthodes SIP, `SUBSCRIBE`, `NOTIFY` et `PUBLISH` fournissent un mécanisme générique de communications asynchrones fondé sur les événements. En plus de la présence que nous venons de voir, il est utilisé pour notifier de l'accomplissement d'un transfert d'appels, de messages en attente, ou d'une saisie en cours au clavier. Ce mécanisme sert également à la signalisation permettant de gérer des conférences.

La téléphonie est une application multimédia gourmande en bande passante malgré la mise en œuvre de techniques de compression. Ceci est particulièrement vrai en visiophonie où la conversation inclus un flux vidéo. Une extension de SIP permet d'effectuer de la réservation de ressources afin de garantir le bon fonctionnement de la communication lorsque celle-ci sera

établie. La réservation de ressources consiste à refuser une connexion s'il ne reste plus assez de bande passante à un instant donnée et à bloquer l'utilisation de bande passante disponible lors de l'initialisation d'une communication. Ce mécanisme garantit ainsi que pour toutes sessions établies, le flux de données associé sera correctement acheminé entre les parties.

Pour faire face à l'augmentation des extensions et des fonctionnalités, il est nécessaire que les clients SIP connaissent les caractéristiques des serveurs qu'ils utilisent et des clients avec lesquels ils communiquent. Par exemple, la description d'un flux en terme de type de données est vitale pour établir l'interopérabilité de deux interlocuteurs et doit prendre en compte les différents codec possibles. La découverte des fonctionnalités est réalisée avec la méthode `OPTIONS`. L'extensibilité ayant été un critère de conception du protocole SIP, cette méthode a été introduite dès le début et fait partie de la spécification de base.

Il est important de retenir qu'il n'y a pas de nouvelle méthode SIP standardisée depuis octobre 2004. Le protocole SIP a désormais atteint un seuil en terme d'expressivité pour la signalisation. Les extensions se focalisent désormais sur des nouvelles charges utiles, des nouveaux en-têtes et des nouvelles valeurs pour les différents paramètres. Ces extensions des charges utiles ont pour but d'intégrer de nouveaux périphériques et de nouvelles fonctionnalités sur un réseau SIP.

3.1.5.4 Services et serveurs d'applications

Afin de fournir les mêmes fonctionnalités que les systèmes de téléphonie patrimoniaux, il est possible de mettre en place des serveurs d'applications SIP (Figure 3.4, à droite) sur lesquels les utilisateurs peuvent activer des fonctions comme une boîte vocale ou une redirection d'appels. L'utilisateur Bob peut par exemple configurer une redirection vers un secrétariat pour les appels qu'il ne peut pas prendre. Il lui suivit d'indiquer l'URI `sip:secretay@labri.fr` dans son service de redirection.

3.2 Services

La téléphonie SIP, avec ses nouvelles extensions, ouvre la voie à une profusion de nouveaux services. Le développement de ces services nécessite cependant des compétences certaines sans lesquelles la fiabilité du système de téléphonie peut être compromise.

3.2.1 Opportunité

Les plates-formes téléphoniques sont historiquement fermées et propriétaires. Dans ce contexte, l'ajout de nouveaux services est long et coûteux. Une fois le choix d'achat d'un commutateur téléphonique (PABX – *Private Automatic Branch eXchange*) réalisé, le client est contraint de s'adresser à son prestataire de service pour toute modification ou activation de services même basiques. Dans le cas de services innovants comme ceux développés dans le cadre d'un Couplage Téléphonie Informatique (CTI) le client doit bien souvent s'adresser au fabriquant du système. Le fabriquant développe alors un service ad hoc et profite de son monopole pour facturer au prix fort ce développement. Le couplage téléphonie informatique restait de fait, soit un concept pour la plupart des entreprises, soit une réalité coûteuse réservée aux grandes entreprises ou aux entreprises spécialisées.

L'apparition d'implémentations de système de téléphonie aux codes sources ouverts renforce la position des utilisateurs. Tous les programmeurs peuvent désormais, moyennant les

compétences techniques nécessaires, ajouter des services à un système de téléphonie. La suppression de la position de monopole des fabriquants de PABX, accompagnée de la simplification du développement des systèmes de téléphonie, favorise l'innovation et le développement des services de téléphonie.

Enfin, le fait que la téléphonie devienne une application sur IP, au même titre que le courrier électronique ou le Web, offre une simplification pour les interactions avec les autres applications. De plus le protocole SIP épouse la structure et la philosophie des autres protocoles textuels sur IP. Par exemple, l'utilisation des URI SIP facilite l'intégration avec le courrier électronique ; il est possible d'envoyer un courrier au destinataire d'un appel n'ayant pas pu répondre. Un éventuel message vocal peut également être mis en pièce jointe.

La convergence des réseaux de télécommunications et des réseaux informatiques offre une opportunité historique pour le développement de services de téléphonie. En effet, les services de téléphonie peuvent désormais aisément exploiter les ressources d'un réseau informatique tel que les serveurs de courrier électronique ou les bases de données. Le processus de développement de systèmes de téléphonie et des services de téléphonie est désormais le même que celui des autres systèmes informatiques. Le développement de services de téléphonie est donc simplifié par rapport à ceux destinés aux PABX patrimoniaux. Ce développement n'est pas pour autant simple et les développeurs doivent posséder un large domaine de compétences.

3.2.2 Développeurs

La simplification du développement des services pour la téléphonie sur IP est une réalité. Toutefois, les domaines de compétences nécessaires de la part d'un développeur restent importants. Il lui faut connaître des domaines aussi variés que les télécommunications, les réseaux informatiques et la programmation distribuée.

L'utilisation de nombreux services, tels que des annuaires, des bases de données, le Web et le courrier électronique, élargit encore le champ des compétences nécessaires au développement de services. Paradoxalement, l'utilisateur final, qui n'a généralement pas ces compétences, est moteur dans l'expression du besoin de services. Son objectif est de faciliter son quotidien. Dans les faits, l'utilisateur final doit passer par un tiers compétent pour le développement de ses services.

En fait, il est intéressant d'élargir la base des développeurs en facilitant le développement de services et en limitant l'expertise nécessaire. Toutefois, l'administration d'un système exécutant du code développé par des tiers ou des utilisateurs finaux pose un certain nombre de problèmes, notamment de sécurité. L'ouverture des plates-formes de téléphonie ne doit pas se faire au détriment de la sécurité du système.

3.2.3 Contraintes

Afin de préserver la fiabilité des systèmes de téléphonie, y compris en présence de services, il est important de bien concevoir les services. Rosenberg et coll. [RLS99] définissent un cahier des charges précisant où doivent être déployés les services dans le réseau [WS00], quel est le cycle de vie des programmes, c'est-à-dire la persistance de l'état des services, quelles doivent être les restrictions d'accès aux ressources du réseau et enfin, quelles sont les possibilités d'interfaces entre un serveur d'applications et la logique d'une application.

3.2.3.1 Emplacement des services

Les services peuvent être déployés soit dans le réseau, sur des serveurs d'applications, soit sur les terminaux. Selon l'option retenue, les possibilités de services diffèrent comme le mentionnent Wu et Schulzrinne [WS00]. Le Tableau 3.1 indique quelques exemples de services et l'emplacement où ils peuvent être déployés. Nous pouvons remarquer deux points importants. D'une part, tous les services utilisant ou manipulant le flux multimédia, à l'exception du service de sonnerie différenciée et du service de transfert, nécessitent d'être déployés sur un terminal ou sur un serveur capable de gérer le média. D'autre part, les services de routage peuvent être déployés sur un serveur dans le réseau. On retrouve dans l'emplacement des services, la dichotomie entre services de routage et services entités que nous avons vue à la section 2.2. Les services entités nécessitent la gestion du média.

Type de services	Service	Emplacement du service		
		Terminal	Cœur de réseau (Serveur)	
			Serveur mandataire	Serveur avec gestion du média
Service de routage	Redirection sur occupation	oui	oui	oui
	Redirection sur non réponse	oui	oui	oui
	Redirection sur absence	non	oui	oui
	Anonymisation	non	oui	oui
Services entités	Sonnerie différenciée	oui	non	non
	Appel en attente	oui	non	oui
	Transfert	oui	non	non
	Conférence	oui	non	oui
	Boîte vocale	oui	non	oui

TAB. 3.1: Services possibles en fonction de leur emplacement

D'autres préoccupations doivent être prises en compte lors de l'analyse du tableau 3.1. Si à première vue, le déploiement de service sur les terminaux semble intéressant, les administrateurs des systèmes d'information ne peuvent pas, pour des raisons de sécurité, leur faire confiance. Il est en effet facile pour un attaquant de connecter un terminal exécutant du code malveillant. Les terminaux ne peuvent donc pas *en général* accéder à toutes les ressources du réseau. Les services tirant parti d'annuaires ou d'agendas partagés sont alors impossibles. De plus, les terminaux ont par essence une connectivité volatile. Par exemple, un service déployé sur le téléphone Wifi d'un usager ne sert à rien si le téléphone est en dehors de la zone de couverture Wifi. Enfin, le déploiement de services dans le réseau présente l'avantage de simplifier leur gestion. Il n'est alors plus nécessaire de gérer l'hétérogénéité des terminaux et le comportement est plus prévisible lorsque l'utilisateur dispose de plusieurs terminaux (par exemple, téléphone filaire, téléphone GSM et ordinateur) pour gérer ses appels.

Bien que le réseau soit l'emplacement idéal pour le déploiement des services de routage, un tel déploiement peut toutefois fragiliser le réseau. Afin de résoudre ce conflit d'intérêts, l'architecture des réseaux SIP prévoit le déploiement des services sur des serveurs dédiés, les serveurs d'applications, comme c'est le cas notamment dans les architecture IMS. Ainsi, en cas d'arrêt d'un ou de plusieurs serveurs d'applications, le fonctionnement nominal du système de téléphonie est préservé. La contrepartie est une latence supplémentaire due aux échanges

réseaux entre le serveur mandataire de cœur et les serveurs d'applications. Cette latence ne pose pas de problème en pratique car la durée perçue par l'usager lors de l'établissement d'un appel ne varie pas.

3.2.3.2 Invocation du programme et persistance de l'état

Tous les services ne nécessitent pas d'être invoqués à chaque événement de signalisation, c'est-à-dire à chaque message réseau ou événement protocolaire tel qu'une alarme *(timer)*. Par exemple, les services de redirection d'appels n'ont besoin que de la méthode `INVITE` pour s'activer. Ce type de service est dit *sans état (stateless)*. Pour un service de redirection sur non réponse, il est nécessaire d'avoir une réponse négative ou aucune réponse de la part de l'appelé. Une fois la réponse envoyée à l'appelant, l'exécution peut s'arrêter. On parle de service *avec état associé à une transaction (transaction stateful)* car un état est maintenu durant l'ensemble de la transaction `INVITE`.

Il existe cependant de nombreux autres services qui utilisent davantage d'événements de signalisation. Un service enregistrant la durée des appels à des fins statistiques utilise non seulement l'initialisation de l'appel avec la méthode `INVITE`, mais également l'acquittement et la terminaison avec, respectivement, les méthodes `ACK` et `BYE`. Lorsqu'un état est maintenu durant l'ensemble de l'appel, on parle de service *avec état associé à un appel (call stateful)*.

Le type de service développé a un impact sur l'utilisation mémoire, et donc sur le passage à l'échelle d'un service. Il est donc important de minimiser l'état au stricte nécessaire.

3.2.3.3 Accès aux ressources

Les systèmes de téléphonie patrimoniaux sont fiables et le téléphone est désormais une commodité à laquelle les usagers font confiance. Il ne faut pas que le passage à la téléphonie IP et l'ajout de services compromettent la fiabilité du système. La compromission du système peut en effet conduire, d'une simple interruption d'un service d'un utilisateur, à l'arrêt complet et brutal de la plate-forme de téléphonie, empêchant de ce fait toute communication.

L'intégration des fonctionnalités entre les différentes applications disponibles sur le réseau nécessite d'autoriser les services à accéder à des ressources jusque là protégées. L'accès à ces ressources rend le système propice aux attaques et au vol de données. Les services de téléphonie doivent donc faire l'objet d'une validation avant leur déploiement dans le système d'information.

3.2.3.4 Environnement de développement

Le principe d'un service de téléphonie est de manipuler le protocole de signalisation sous-jacent, en l'occurrence le protocole SIP. Cependant, exposer l'ensemble des informations des messages SIP aux développeurs et leur permettre des modifications arbitraires, empêche toute garantie quant au respect du protocole SIP. Il convient donc de définir l'ensemble des modifications possibles permettant de créer un maximum de nouveaux services, tout en préservant la fiabilité du système.

Les services de téléphonie peuvent être plus ou moins génériques. Certains s'appliquent à l'ensemble d'une entreprise ou d'un groupe dans l'entreprise, tandis que d'autres ne s'appliquent qu'à un individu. Dans les deux cas, il est intéressant de réutiliser du code développé dans d'autres systèmes de téléphonie. Il devient alors possible de déployer dans d'autres entreprises les différents services, indépendamment du système téléphonique de celles-ci. La

portabilité des services est un avantage pour les sociétés de services en ingénierie informatique dans le cas de services génériques. Elles peuvent ainsi factoriser leur coût de développement entre plusieurs clients. La portabilité des services est également un avantage pour l'usager qui peut ainsi déployer son service à son domicile, chez son opérateur et à son travail.

Il est important de fournir aux développeurs un environnement de développement adapté. Celui-ci doit leur permettre de se concentrer sur ce que doit faire un service, plutôt que comment il le fait. De plus, un environnement adapté doit faciliter, voire même automatiser, la réutilisation de l'existant.

3.3 Bilan

Après des décennies de séparation, les réseaux de télécommunications et les réseaux informatiques ont maintenant convergé. Cette convergence a permis la mutualisation des réseaux physiques et donc une réduction des coûts. Un autre effet a été le développement de protocoles de téléphonie IP afin d'assurer la continuité de service entre les réseaux. La téléphonie IP en général, et le protocole SIP en particulier, offre un nouveau paradigme aux communications sur les réseaux informatiques au moins aussi révolutionnaire que les protocoles SMTP et HTTP en leur temps.

Une fois la plate-forme de téléphonie fonctionnelle, sa valeur ajoutée provient de l'ajout de services. La difficulté de développement de services dans les systèmes patrimoniaux de téléphonie est pourtant un frein important à l'apparition de services innovants. L'ouverture des plates-formes de téléphonie a simplifié ce développement. Il n'en demeure pas moins encore complexe et de nombreux défis doivent être relevés par des développeurs aux compétences multiples.

Afin, de garantir la *fiabilité* du système, il est important de *vérifier* les services avant leur déploiement dans le système. Toutefois, la fiabilité ne doit pas être atteinte au détriment des *fonctionnalités* fournies par le langage. Un compromis doit être trouvé. Nous proposons de mettre en œuvre l'approche des langages dédiés afin de déterminer ce compromis.

Chapitre 4

Langages dédiés

L'ingénierie logicielle consiste à concevoir et développer des programmes informatiques. De nos jours, les programmes développés couvrent un large spectre de problèmes. Les problèmes peuvent être regroupés en différents domaines. On peut alors s'intéresser à la définition d'un langage dédié (en anglais DSL pour *Domain-Specific Language*) pour la résolution des problèmes d'un domaine particulier. Parmi les domaines ayant déjà fait l'objet d'une étude conduisant à la création d'un langage dédié, on trouve la finance [vD97], les systèmes d'exploitation [LMB02, MRC+00, Thi98], les protocoles réseaux applicatifs [Bur08], les bases de données [KT03], les documents structurés [MGB+04, W3C02], les images [Ker82], la musique [Lan90], la chimie [Ben86], les services Web [ABBC99, BMS02, CM02], ou encore la conception matérielle [Ash02]. Enfin, l'étude et le développement des langages en général, et des langages dédiés en particulier, a donnée lieu à plusieurs langages dédiés tels que BNF [Knu64], SDF [HHKR89], Lex [LS75], Yacc [Joh75].

Ce chapitre présente succinctement plusieurs méthodologies et outils associés qui sont disponibles pour faciliter la création de langages dédiés [CM98, BPM04, BKVV08]. La première étape de la conception d'un langage dédié nécessite de délimiter et d'analyser le domaine auquel il est dédié. À la suite de l'analyse d'un domaine, il est nécessaire de définir les objectifs du DSL. Ces objectifs peuvent être consignés dans un cahier des charges. Un langage est ensuite défini en respectant les contraintes du domaine et les objectifs des concepteurs. La définition d'un langage comprend une syntaxe et une sémantique. Ces éléments permettent de réaliser des analyses sur les programmes écrits à l'aide du DSL afin d'identifier les programmes valides. Après avoir été défini, le langage est implémenté. L'implémentation d'un DSL comporte généralement un analyseur, un compilateur ou un interprète et éventuellement un environnement d'exécution.

4.1 Analyse de domaine

L'évolution d'un programme en fonction des nouveaux besoins, de modifications de l'environnement d'exécution ou d'améliorations conduit à de multiples versions. Pour limiter le coût de développement global de ces versions, Parnas [Par76] propose de s'intéresser d'abord à un ensemble de programmes apparentés qu'il nomme *famille de programmes*. Un ensemble de programmes forme une famille « *lorsqu'il est plus utile d'étudier les programmes de cet ensemble en étudiant d'abord leurs propriétés communes puis en déterminant les propriétés spécifiques de chaque membre de la famille* ».

Le concept de famille de programmes a depuis été repris pour définir un domaine. Le terme d'*analyse de domaine* a été introduit par Neighbors [Nei80] comme « *l'activité d'identification des objets et opérations d'une classe de systèmes similaires* (une famille de programmes) *dans un domaine particulier de problèmes* ». La conception d'un langage dédié à un domaine est guidée par l'idée que la spécialisation d'un langage, pour manipuler les objets et opérations d'un domaine, permettra de faciliter le développement de nouveaux membres de la famille associée à ce domaine. Czarnecki et Eisenecker [CE00] précisent la définition d'un domaine comme étant une sphère de connaissances. « *Cette sphère de connaissances s'étend jusqu'à maximiser la satisfaction des besoins des intervenants. Elle inclut l'ensemble des concepts et de la terminologie utilisés par les experts de cette spécialité ainsi que les connaissances nécessaires au développement de systèmes logiciels dans cette spécialité* ». Malgré cette définition plus précise, lorsque le domaine est vaste, les experts le décomposent en plusieurs sous-domaines. Prieto-Díaz [PD90] définit un domaine « *comme un réseau dans des structures semi-hiérarchiques* ». En bas de la hiérarchie, se trouveraient de petits domaines contenant des primitives comme des langages assembleurs et les opérations arithmétiques, et en haut de la hiérarchie, les domaines plus larges et plus complexes. La complexité d'un domaine est alors liée au nombre de sous domaines nécessaires pour le définir.

L'étude d'un domaine est la pierre angulaire de la construction d'un DSL. De nombreuses méthodologies ont été développées pour caractériser et modéliser les domaines telles que FODA *(Feature- Oriented Domain Analysis)* [KCH+90], FAST *(Family-Oriented Abstractions, Specification, and Translation)* [Wei98, Wei96], DARE *(Domain Analysis and Reuse Environment)* [FPDF98], DSSA *(Domain- Specific Software Architectures)* [Tra95] ou ODE *(Ontology-based Domain Engineering)* [dAFGD02] pour n'en donner que quelques unes. Des outils permettent d'assister l'analyste de domaines dans la formalisation d'un domaine. Nous pouvons par exemple citer DOMAIN [TTC95], DARE [FPDF98] ou FDL [vDK02].

L'analyse de domaine n'est pas réservée à la conception de DSL, elle est également utilisée lors de la conception de lignes de produits. En effet, bien qu'il s'agisse de finalités différentes comme la conception de lignes de produits ou la conception de langages dédiés, les approches présentées partagent le même principe général : elles sont basées sur des informations provenant d'experts et d'utilisateurs d'un domaine. Ces informations sont triées en deux catégories : d'une part, les points communs entre les membres d'une famille de programmes dans un domaine donné, et d'autre part les variations entre ces membres. On parle d'analyse de points communs.

4.1.1 Sources d'informations

La définition d'un domaine commence par une analyse des besoins. Dans le cas d'un langage, il faut aussi que les concepteurs du langage s'approprient la terminologie du domaine étudié. Une analyse précise des besoins permet d'éviter l'écueil de la surconception, qui consiste à ajouter des fonctionnalités conceptuellement intéressantes mais ne présentant aucune utilité dans la pratique. L'appropriation du jargon d'un domaine par les concepteurs du langage leur permet de définir les concepts et la terminologie propres au domaine [Wil04].

Les sources d'informations sont liées à des échanges entre les experts d'un domaine et les experts langages. Le processus de définition du domaine est souvent basé sur des échanges en langue naturelle sous forme écrite ou orale. Il s'agit donc d'un processus où le facteur humain est important. La définition d'un domaine peut être longue et comporter des lacunes. La définition d'un domaine se fait en pratique conjointement avec la conception du langage lors d'un processus itératif [Wei98, dAMC+06]. Ce processus permet de valider l'analyse de

domaine.

En l'absence d'un DSL, des programmes écrits à l'aide de langages généralistes, sont mis en œuvre pour résoudre les problèmes du domaine. Les programmes ainsi produits constituent une base de connaissances qui peut être exploitée. Ces programmes sont des membres de la famille de programmes que cible le futur DSL. L'étude de ces programmes et leur comparaison permettent d'en définir les points communs et les variations.

4.1.2 Points communs

L'objectif de l'analyse de domaine consiste entre autre à définir les points communs d'une famille de programmes. Weiss [Wei98] définit les points communs comme « *la liste des postulats qui sont vrais pour tous les membres de la famille* ».

Parnas [Par76] propose un processus de développement fondé sur des « *décisions abstraites* ». Il consiste à concevoir de façon abstraite l'architecture d'un système par raffinements successifs. À chaque étape, le système est un peu plus défini jusqu'à obtenir un système complet. Un système complet est fonctionnel et peut donc être exécuté. Une nouvelle version du système abstrait est créée à chaque nouvelle alternative possible lors de la conception. Lors d'un tel processus de développement, l'analyse de points communs doit définir le plus récent ancêtre commun de la famille. Le processus de développement proposé par Parnas est illustré à la figure 4.1. Le plus récent ancêtre commun dans l'illustration présentée est celui où le système est défini à 33% (troisième nœud).

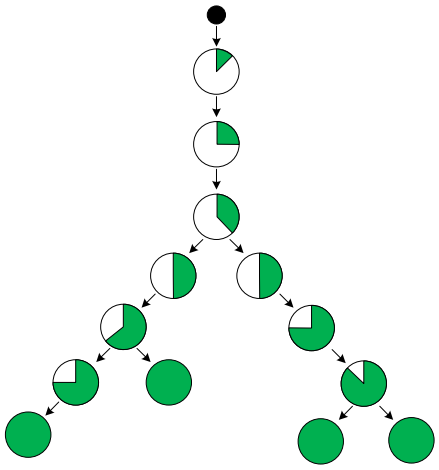


FIG. 4.1: Développement de famille de programmes par décisions abstraites [Par76] (illustration adapté de [Rév01])

Les nœuds fils de cet ancêtre présentent tous des variations entre eux, y compris les feuilles représentant les membres fonctionnels de la famille. Une fois les points communs d'une famille de programmes identifiés, l'analyse de domaine s'intéresse aux variations que présentent les membres de cette famille.

4.1.3 Variations

L'étude des variations des membres d'une famille est de deux types. Il faut d'une part identifier les variations de fonctionnalités et d'autre part identifier les paramètres de variation. L'étude de ces variations définit l'expressivité du futur DSL.

4.1.3.1 Variations de fonctionnalités

L'étude des variations de fonctionnalités correspond à l'étude des fonctionnalités dans les lignes de produits [vDK02]. Un diagramme graphique de fonctionnalités *(Graphical Feature Diagram)* peut être mis en œuvre pour définir les différentes fonctionnalités d'une ligne de produits. Il est possible d'appliquer cet outil pour caractériser les variations entre les membres d'une famille de programme.

La figure 4.2 illustre l'application de cet outil sur le cas d'une voiture [vDK02]. Une notation graphique permet d'exprimer diverses possibilités : les fonctionnalités optionnelles (rond vide), les fonctionnalités à choix unique (triangle vide) et les fonctionnalités à choix multiple (triangle plein). Il est possible d'utiliser des outils tels que FeatureIDE [LAMS05] fournissant du support aux utilisateurs pour réaliser ce type de diagramme.

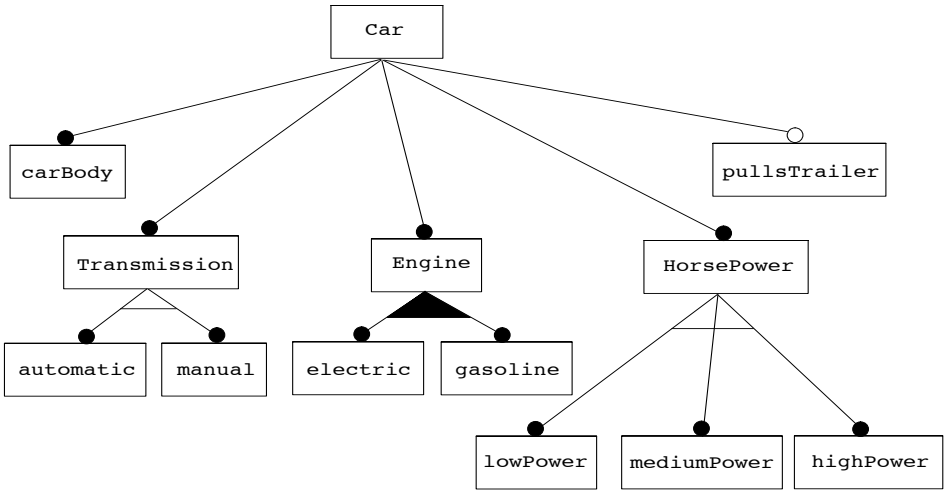


FIG. 4.2: Exemple de représentation graphique pour les fonctionnalités d'une voiture

La représentation graphique présente l'avantage d'être facile et rapide à lire par un humain. Elle fournit un support aux discussions entre les experts lors de la conception d'un DSL. Afin, de pouvoir contrôler automatiquement la validité de modèles plus complexes, une représentation textuelle équivalente est utilisée comme l'illustre la figure 4.3. van Deursen et Klint [vDK02] proposent des règles de réécriture permettant de normaliser, d'étendre et de contraindre une description de fonctionnalités. Ils proposent d'appliquer ces règles afin de générer une description UML *(Unified Modeling Language)* [UML] puis une implémentation abstraite d'un système en Java. Ils concluent sur l'utilisation de la description de fonctionnalités pour définir la grammaire d'un DSL. Un générateur pour ce DSL pourrait alors produire du code utilisant une instance du système généré via la description UML.

```
1 Car: all( carBody, Transmission, Engine, HorsePower, pullsTrailer? )
2 Transmission: one-of( automatic, manual )
3 Engine: more-of( electric, gasoline )
4 HorsePower: one-of(lowPower, mediumPower, highPower)
```

FIG. 4.3: Exemple de représentation textuelle pour les fonctionnalités d'une voiture

4.1.3.2 Paramètres de variation

Les variations de fonctionnalités correspondent soit à des opérations qui peuvent être invoquées ou non par le développeur, soit à des motifs de code [GHJV95] que le développeur met en œuvre. Dans les deux cas, il est fréquent qu'une partie varie en fonction du membre de la famille que le développeur programme. Cette partie variable correspond, soit à des paramètres des opérations invoquées, soit à des paramètres d'un motif de code.

L'étude de ces paramètres de variation est liée à la formalisation des concepts et des objets d'un domaine. Dans l'exemple présentant les fonctionnalités d'une voiture, il est possible de préciser, par exemple, le nombre de rapports de la transmission, ou l'autonomie d'énergie, c'est-à-dire la capacité des accumulateurs ou la taille du réservoir selon le mode d'alimentation de la voiture, électrique ou essence.

Il est important de bien caractériser les paramètres de variation, notamment leur type et leur domaine de valeurs. Par exemple, la taille d'un réservoir peut être définie par un entier naturel dans un intervalle borné et être exprimée en litre ou en gallon.

Le dernier aspect des paramètres de variation est l'instant où leur valeur est connue. On parle de temps de liaison. Les paramètres de variation sont ainsi définis plus ou moins tôt dans le cycle de vie d'une application : à la conception, au déploiement ou à l'exécution. Plus la valeur d'un paramètre est connue tard, plus une valeur erronée aura un impact important car elle aura fait l'objet de moins de vérification et aura moins de chance d'être détectée. Une valeur erronée à l'exécution peut, par exemple, causer l'arrêt inopiné d'une application.

L'analyse d'un domaine est une phase cruciale pour la conception d'un DSL. C'est lors de cette analyse que tous les concepts et la terminologie d'un domaine sont explicités. Ils seront ensuite repris lors de la conception du DSL. Il existe cependant d'autres facteurs ayant un impact sur la conception d'un DSL. Ces autres facteurs peuvent être exprimés sous la forme d'un cahier des charges que doivent respecter les concepteurs du DSL.

4.2 Définition d'un cahier des charges

Lors de la conception d'un DSL, il est primordial de motiver la conception du DSL. Quel est le besoin de ses futurs utilisateurs ? Quelles sont les difficultés du développement logiciel avec les solutions actuelles dans le domaine considéré ? L'utilisation de l'approche DSL peut également comporter des risques. Il convient de les connaître et d'en mesurer l'impact.

Pour répondre à ces questions et évaluer les risques, une approche possible consiste à définir un cahier des charges pour la conception d'un DSL. Ce cahier des charges exprime les objectifs que doit atteindre le DSL. La définition de ces objectifs est propre à chaque DSL. Elle doit permettre de réaliser les arbitrages qui auront lieu lors du développement du DSL. Les objectifs sont de deux ordres : d'une part les objectifs relatifs aux risques de développement

et d'utilisation des DSL, et d'autre part les objectifs relatifs aux gains induits par l'utilisation d'un DSL.

4.2.1 Risques potentiels

L'approche des DSL souffre de problèmes ou de risques chroniques : les coûts initiaux, liés au développement et au déploiement du DSL, la maintenance, liée à l'évolution du DSL, la communauté des utilisateurs, et enfin la performance des programmes.

4.2.1.1 Coûts initiaux

L'analyse de domaine et le développement d'un DSL sont consommateurs de ressources humaines et de temps. Il y a donc un coût financier associé à la création d'un DSL. Toutefois, l'expérience montre que le surcoût initial peut être rapidement amorti. Dans le cas de l'utilisation de la méthode FAST [Wei98, Wei96], le coût de production d'un nouveau membre de la famille est réduit d'environ quatre fois et le surcoût initial lié à l'analyse du domaine est amorti au troisième membre de la famille développé. Cette étude se cantonne à l'analyse de domaine et ne concerne pas le développement d'un DSL. Toutefois d'autres études telles que la comparaison réalisée par van Deursen [vD97] montre les avantages à disposer d'un DSL pour accéder à un intergiciel.

Hudak [Hud96] propose une approche permettant de limiter le coût initial. Cette approche est fondée sur les langages dédiés enchâssés. Le principe est de profiter de l'infrastructure et des outils de l'environnement de développement d'un langage généraliste.

Un DSL a un cycle de vie similaire aux autres approches en génie logiciel. Un fois conçu, il doit être maintenu et évoluer au gré des besoins qui n'ont pas été identifiés lors de l'analyse de domaine.

4.2.1.2 Maintenance d'un DSL

Si pour la conception initiale, il paraît évident qu'un expert du domaine et un spécialiste langage doivent être impliqués, leur présence durant la maintenance d'un DSL et de ses outils (c.-à-d. éditeur, débogueur, compilateur, interprète) n'est malheureusement pas souvent planifiée. Les compétences nécessaires à la maintenance d'un DSL doivent être prises en compte au même titre que les autres compétences très spécifiques qui peuvent être nécessaires dans une entreprise. La maintenance d'un DSL est alors à un problème de gestion des ressources humaines au sein d'une entreprise. De plus, il est possible d'externaliser les compétences nécessaires et de faire appel à des sociétés de service en ingénierie informatique, par exemple. Il faut alors disposer des documents de travail utilisés et produits lors de l'analyse de domaine afin de ne pas avoir à supporter une seconde fois le coût de l'analyse de domaine.

4.2.1.3 Utilisateurs

Un autre obstacle significatif à l'utilisation d'un DSL est lié à sa communauté d'utilisateurs. L'adoption d'un DSL par la plus large communauté possible est un facteur décisif pour le succès d'un DSL. L'adoption d'un DSL dépend principalement de deux points : les solutions alternatives existantes et la facilité d'apprentissage du nouveau langage.

Dans le cas du développement de services Web, le langage PHP [LTM06] est apparu très tôt et est devenu un standard *de facto*. Il possède désormais une large communauté d'utilisateurs

et de nombreuses bibliothèques de fonctions. Bien que des langages dédiés [ABBC99, BMS02, CM02] proposent des solutions plus sûres et plus efficaces pour le développement de services Web, leur utilisation reste anecdotique.

Pour qu'une communauté se crée autour d'un langage, il faut que celui-ci soit facile d'accès. Étant dédiés à un domaine, les DSL ont sur ce point l'avantage d'être *généralement* plus petits en terme de concepts. Un nouvel utilisateur a certes besoin d'un certain temps d'adaptation et d'apprentissage mais il est rapidement productif [SKG+07]. L'utilisation des concepts et de la syntaxe d'un domaine, dont les utilisateurs sont déjà familiers, est également un avantage indéniable. Enfin, l'implication des utilisateurs, dès l'analyse de domaine, est un facteur déterminant pour le succès d'un DSL [Wil04].

4.2.1.4 Performances

Une des critiques dont les DSL sont la cible est que les programmes ainsi produits seraient moins performant que leurs homologues écrits à l'aide de langages généralistes. Dans la pratique, les deux approches utilisent généralement le même intergiciel dédié au domaine. De plus, l'analyse d'abstractions haut niveau permet aux compilateurs de systématiser des optimisations indépendamment du niveau de compétence d'un développeur. Bien qu'il n'existe pas d'étude théorique sur le sujet, de nombreuses expériences montrent empiriquement que les performances sont comparables [MP99, Thi98, RM01, BRLM07, Bur08].

L'idée qu'un DSL est moins performant provient de l'usage fréquent d'un interprète pour les programmes du langage plutôt qu'un compilateur. En effet, le développement d'un interprète est plus simple et plus rapide que celui d'un compilateur pour un même langage. Toutefois, le surcoût d'un interprète à l'exécution peut être supprimé en utilisant par exemple l'approche Sprint [Thi98, CM98].

Malgré le coût initial et le risque lié à la maintenance, l'approche des langages dédiés présente des avantages qui en font une solution attractive. Nous allons à présent voir les principaux avantages de cette approche.

4.2.2 Avantages escomptés

Le large spectre de domaines auquel l'approche a été appliquée, montre empiriquement que l'utilisation d'un DSL est une plus value pour les usagers. Plusieurs facteurs expliquent l'intérêt pour les DSL. Tout d'abord, ils permettent un développement plus simple et plus rapide qu'une approche généraliste pour résoudre des problèmes d'un domaine. De plus, ils systématisent la réutilisation de l'intergiciel sur lequel ils peuvent être fondés ainsi que les bonnes pratiques de conception d'un domaine. Enfin, grâce aux abstractions qu'ils fournissent aux développeurs, ils rendent les programmes développés plus fiables car vérifiables automatiquement.

4.2.2.1 Programmation plus facile

L'utilisation des DSL, dans des domaines très variés, comme la finance [vD97], la musique [Lan90], les images [Ker82] ou la conception de circuits électroniques [Ash02], induit que les utilisateurs ne sont pas forcément des programmeurs informatiques. Il faut donc que le langage, s'adapte à leur expertise. Pour ces utilisateurs, l'utilisation de l'outil informatique n'est qu'un moyen pour effectuer plus facilement une tâche liée à leur profession ; l'adoption d'un DSL en est donc facilité. Les utilisateurs peuvent alors se concentrer sur leur métier

et résoudre leurs problèmes plus simplement. Par exemple, grâce au DSL Eon [SKG+07], les programmeurs d'applications embarquées sont à même de développer environ quatre fois plus vite que leurs homologues utilisant le langage C. Le DSL Risla, dans le domaine de la finance, permet de réduire la durée des projets, d'environ trois mois à au plus trois semaines [vD97].

4.2.2.2 Réutilisation systématique

L'analyse de domaine est un prérequis à la conception d'un DSL et constitue une valeur ajoutée. Elle peut être utilisée par les futurs experts du domaine qu'elle décrit ainsi que par les nouveaux utilisateurs. Ces deux types d'intervenants profitent ainsi de l'effort de conceptualisation qui a été fait. Il leur est ainsi plus simple et donc plus rapide d'acquérir une expertise dans un domaine donné.

L'utilisation d'un DSL favorise intrinsèquement la réutilisation de code. En effet, la réutilisation de code (par exemple, intergiciels, bibliothèques de fonctions) n'est plus à la charge du développeur mais de la responsabilité des outils implémentant le DSL. La réutilisation est alors systématique et indépendante du niveau d'expertise du développeur.

Grâce aux abstractions fournies par un DSL, les programmes développés sont généralement plus concis que leurs homologues écrits avec des langages généralistes. De plus, ils appartiennent à la même famille de programmes. Il est donc plus aisé de dériver un programme existant en un nouveau membre de la famille. La réutilisation de code est donc également facilitée au sein de la famille de programmes.

4.2.2.3 Fiabilité améliorée

À cause de la généricité des langages généralistes, nombre de propriétés sont indécidables. Les DSL introduisent des abstractions et des restrictions qui rendent décidables des propriétés spécifiques à un domaine. Ces propriétés peuvent alors être vérifiées automatiquement par des outils dédiés au langage. Il est ainsi possible de détecter des programmes ne respectant pas des contraintes du domaine. Par exemple, le langage <BigWig> [BMS02] sert au développement de site Web. Pour qu'un programme <BigWig> soit valide, il faut qu'il produise une page HTML valide, c'est-à-dire respectant un enchevêtrement correcte de balises HTML. Cette contrainte est vérifiée par le compilateur. Ainsi seuls les services valides sont déployés sur le serveur Web. Une telle garantie est impossible en utilisant un langage généraliste comme PHP [LTM06].

Enfin, l'utilisation d'abstractions de haut niveau permet au développeur de ne pas avoir à se soucier des détails d'implémentation. Par exemple, l'utilisation de langages tels que Bossa ou Devil [LMB02, RM01], dédiés respectivement à l'ordonnancement de processus et au développement de pilotes de périphériques, permet de s'affranchir de développements C particulièrement bas-niveau. De tels développements sont des sources d'erreurs non négligeables alors qu'ils ont un impact sur la stabilité du système car ils s'agit de programmes s'exécutant avec les privilèges du noyau dans le système d'exploitation.

4.3 Définition d'un langage dédié

Une fois l'analyse d'un domaine réalisée et les objectifs du DSL spécifiés dans un cahier des charges, les experts langages définissent, en coopération avec les experts du domaine, le langage dédié. Afin de définir un langage, une méthodologie éprouvée et faisant consensus

consiste à le formaliser. Cette formalisation se décompose en plusieurs phases : la définition de la syntaxe du langage, la sémantique statique et la sémantique dynamique.

4.3.1 Syntaxe

La *syntaxe* des langages informatiques est définie à l'aide de grammaires. Il existe plusieurs notations pour représenter une syntaxe. Les plus répandues sont la BNF, *Backus-Naur Form* [Knu64], et SDF, *Syntax Definition Formalism* [HHKR89]. Le principe est de décrire la grammaire sous la forme de règles de dérivation exprimant les variations du domaine. La figure 4.4 illustre l'utilisation de la notation BNF sur un langage de manipulation d'entiers et de chaînes de caractères que nous nommerons *Tiny*. Une règle comporte deux types d'éléments : les terminaux et les non-terminaux. Les premiers sont soit des mots clés du langage (par exemple, `compute`, `length` ou les opérateurs mathématiques) soit des paramètres de variation tels que des chaînes de caractères, des entiers, des identifiants. Les terminaux sont déterminés par la terminologie identifiée lors de l'analyse de domaine. Les seconds éléments, les non-terminaux, référencent les règles de production de la grammaire et définissent les dérivations possibles par une combinaison de terminaux et de non-terminaux.

program ::= `compute` *expression*
expression ::= *integer*
| *string*
| `(`*expression*`)`
| *expression* `+` *expression*
| *expression* `-` *expression*
| *expression* `*` *expression*
| *expression* `/` *expression*
| `length` *expression*
| `substr` *expression* *expression* *expression*

FIG. 4.4: Notation BNF du langage *Tiny*

Un programme est syntaxiquement valide s'il existe un arbre de dérivations[1] respectant les règles de la grammaire. Le programme de la figure 4.5 est ainsi un programme valide dont le résultat est la chaîne `"Hello"`. Afin de construire l'arbre décrivant un programme, des outils génératifs tels que Lex [LS75] et Yacc [Joh75] ou Stratego/XT [BTKVV06], peuvent être utilisés pour produire un programme réalisant l'analyse lexicale et syntaxique de programmes vis à vis d'une grammaire.

```
compute length substr 1 (2*3) "Hello world!"
```

FIG. 4.5: Exemple de programme valide pour la grammaire du langage *Tiny*

[1] L'unicité de l'arbre est obtenue grâce à une grammaire non ambiguë. On utilise pour cela des règles d'associativité et de précédence.

4.3.2 Sémantique statique

L'analyse syntaxique n'est toutefois pas suffisante pour dire qu'un programme est valide. Si l'on considère que l'opérateur + du langage *Tiny* ne s'applique qu'à deux entiers, le programme de la figure 4.6 est alors invalide bien que syntaxiquement correcte.

```
compute "Hello " + 1
```

FIG. 4.6: Exemple de programme invalide syntaxiquement correct pour le langage *Tiny*

Un aspect important des langages est de formellement définir le sens des constructions proposées. Afin de définir la sémantique d'un langage, différentes notations peuvent être utilisées. On trouve, de la plus concrète à la plus abstraite, la notation opérationnelle [Plo81], dénotationnelle [Sch86] et axiomatique [Hoa69]. La sémantique axiomatique n'est pas adaptée pour définir complètement la signification d'un programme contrairement aux sémantiques opérationnelle et dénotationnelle. Elle peut être utilisée pour l'analyse de propriétés sur des programmes. La sémantique opérationnelle présente l'avantage de définir de façon précise la signification d'un langage. Elle peut être une étape entre la définition d'une sémantique dénotationnelle et son implémentation [Sch86]. Nous nous intéressons dans la suite de ce document uniquement à la sémantique opérationnelle. Il sera ainsi plus rapide de produire une implémentation.

La sémantique opérationnelle est fondée sur un système de réécriture illustré par la figure 4.7. Lorsque un motif *pattern*, représentant une construction syntaxique d'un langage, est rencontré lors du parcours de l'arbre syntaxique, les propositions *conditions* sont évaluées en considérant l'environnement *env*. Si toutes les propositions sont valides, alors l'environnement env' est produit.

$$\frac{conditions}{env \vdash pattern \rightarrow env'}$$

FIG. 4.7: Règle de réécriture en sémantique opérationnelle

Si l'on considère la sémantique statique de l'opérateur + dans le langage *Tiny*, la règle, Figure 4.8, permet de définir que l'opérateur + ne fonctionne qu'avec des entiers, dénotés par le type Integer. Cette règle indique donc que l'expression représentée par l'opérateur est de type entier si et seulement si les deux expressions utilisées avec l'opérateur sont de type entier. Cette règle ne peut être satisfaite si l'une des expressions est une chaîne de caractères et le programme de la figure 4.6 est ainsi rejeté avant qu'il ne soit exécuté.

Nous venons de présenter la vérification de types à l'aide d'une sémantique statique. Il est également possible de définir d'autres sémantiques statiques permettant d'analyser d'autres propriétés des programmes. Lors de l'analyse de domaine, il est important de définir quelles sont les propriétés importantes, voire critiques, du domaine. En effet, les propriétés retenues ont un impact sur la structure du DSL afin que leur analyse conduise à un résultat décidable.

$$\frac{\begin{array}{c}\vdash expression_1 \rightarrow \mathsf{Integer} \\ \vdash expression_2 \rightarrow \mathsf{Integer}\end{array}}{\vdash expression_1 + expression_2 \rightarrow \mathsf{Integer}}$$

FIG. 4.8: Règle de typage pour l'opérateur + en *Tiny*

4.3.3 Sémantique dynamique

Nous venons de voir comment formellement définir le sens statique des constructions afin de valider un programme par rapport à certaines propriétés comme le typage. Un programme a vocation à être exécuté. Il est donc important de définir également son comportement à l'exécution. Le comportement global d'un programme peut être défini en fonction de chacune des constructions du langage. Enfin, le comportement de chaque construction peut être défini de manière plus ou moins détaillée. On peut par exemple expliciter l'ordre d'évaluation des paramètres d'une fonction. On parle alors de sémantique à petits pas par opposition à une sémantique à grands pas où l'évaluation des paramètres est atomique.

La sémantique opérationnelle définit les opérations réalisées par une construction d'un langage. Dans le langage *Tiny*, l'opérateur + peut ainsi être défini comme produisant l'addition de ces deux opérandes, $value_1$ et $value_2$. La figure 4.9 illustre la sémantique dynamique de l'addition de deux entiers à l'aide d'une sémantique à grands pas.

$$\frac{\begin{array}{c}\vdash value_1 \rightarrow v_1 \\ \vdash value_2 \rightarrow v_2 \\ value := v_1 + v_2\end{array}}{\vdash value_1 + value_2 \rightarrow value}$$

FIG. 4.9: Règle pour l'opérateur + en *Tiny*

L'avantage de formaliser la sémantique opérationnelle réside dans le fait que le développement d'un interprète est alors direct. De plus, l'effort consenti à produire une sémantique indépendante d'une technologie est également récompensé lorsqu'une nouvelle implémentation du langage doit être réalisée dans un nouvel environnement logiciel.

4.4 Implémentation

La dernière étape de conception d'un DSL est son implémentation. Plusieurs systèmes ont été proposés pour faciliter la tâche des développeurs. Ils reposent *généralement* sur la transformation de programmes. C'est notamment le cas des approches DMS (*Design Maintenance System*) [BPM04], Stratego/XT [BKVV08] ou Sprint [CM98]. Elles diffèrent cependant dans leur fonctionnement. Les deux premières utilisent des règles de réécriture exprimées par le développeur du DSL. La méthodologie Sprint propose quant à elle de développer un interprète fondé sur une machine abstraite. L'utilisation d'un interprète est plus flexible car plus simple à développer et à maintenir. Cependant, l'utilisation d'un compilateur est souvent plus

performante. Pour offrir une approche à la fois flexible et performante, la méthodologie Sprint utilise l'évaluation partielle afin de spécialiser l'interprète pour un programme donné, offrant ainsi à la fois la flexibilité d'un interprète et les performances d'un compilateur [Thi98].

Nous avons vu, lors de l'analyse de domaine, que celle-ci utilise des systèmes existants ou conduit à la définition d'un nouveau système tel que décrit par les points communs et les variations identifiés. L'analyse de domaine spécifie une couche d'abstraction [Con04]. Celle-ci peut être implémentée, selon la complexité du domaine, sous la forme d'une ou plusieurs bibliothèques de fonctions, d'un intergiciel ou d'une machine virtuelle. Les outils d'un langage dédié n'ont alors plus qu'à transposer les constructions du langage vers les abstractions fournies.

4.4.1 Couche d'abstraction

Une couche d'abstraction permet de s'affranchir des détails des technologies sous-jacentes. Ainsi, elle masque aux développeurs qui l'utilisent les préoccupations liées au réseau ou à un protocole réseau, par exemple. Selon le domaine d'un système informatique auquel un DSL est dédié, cette couche est plus ou moins importante et plus ou moins complexe. Ainsi dans le langage Devil [Rév01], la couche d'abstraction sous-jacente est quasi-inexistante puisque celui-ci sert au développement de pilotes de périphériques et la majeure partie de la couche basse des pilotes est générée à partir de la spécification Devil. À l'opposé, le langage VisuCom [Lat07], utilisé pour le routage d'appels téléphoniques, repose sur un serveur d'applications de téléphonie IP. Ce serveur d'applications est lui-même fondé sur une pile protocolaire SIP. Consel [Con04] présente une méthodologie pour passer d'une famille de programme à une bibliothèque de fonctions puis d'une bibliothèque à une machine abstraite. Les outils permettant l'exécution des programmes écrits à l'aide d'un DSL n'ont alors plus qu'à le transformer en un programme écrit avec un langage généraliste. Le programme ainsi généré consiste en une succession d'appels aux primitives que la machine abstraite fournit. L'analyse de domaine a un rôle crucial dans les opérations que fournit la machine abstraite. Toute erreur lors de l'analyse de domaine peut avoir un impact négatif sur la machine abstraite, la rendant éventuellement inutilisable. Afin d'éviter cet écueil, il est important de réaliser l'analyse de domaine de façon itérative. Une bonne analyse de domaine conduira alors à une machine abstraite offrant des abstractions idéales pour résoudre les problèmes du domaine.

4.4.2 Outils associés

Un langage informatique présente peu d'intérêt s'il n'est pas associé à des outils. Ces outils peuvent être de nature différente : éditeur, analyseur, visionneur, compilateur ou interprète. Certains de ces outils peuvent être développés pour tous les types de DSL (par exemple, éditeurs) tandis que d'autres comme les compilateurs et les interprètes sont réservés aux DSL exécutables. Dans le contexte du développement de services de communications, nous nous limitons ici à la description des approches permettant d'exécuter des services : l'interprétation et la compilation.

4.4.2.1 Interprétation

L'interprétation est l'implémentation la plus simple d'un langage. Le développement d'un interprète peut être aisément réalisé une fois la sémantique dynamique définie. De plus, tester un interprète est facilité par le fait qu'il produit un résultat immédiat lors du traitement d'un programme.

L'architecture d'un interprète peut rendre encore plus flexible cette approche. Par exemple, l'utilisation de monades [Hud98, ADM05] permet de modulariser le développement de l'interprète, favorisant ainsi les modifications et l'ajout de constructions dans le langage a posteriori améliorant ainsi la maintenabilité.

La souplesse de développement d'un interprète facilite donc la mise au point d'un langage. Cette souplesse est particulièrement intéressante lors du développement d'un DSL. Le développement itératif d'un DSL implique en effet des changements dans la syntaxe ou la sémantique. Ces changements sont validés en les répercutant dans l'interprète.

L'inconvénient majeur de cette approche est le surcoût de l'interprétation. Afin d'éviter ce surcoût structurel, il est nécessaire d'utiliser une approche par compilation.

4.4.2.2 Compilation

Un compilateur transforme un programme d'un langage haut niveau vers un langage présentant des abstractions moins riches. Par exemple, lors de la compilation d'un programme écrit en C vers du code machine, le compilateur réécrit le programme avec des opérations disponibles sur un processeur particulier. De manière similaire, un compilateur Java produit du code binaire pour une machine virtuelle Java à partir de code source.

Dans le cas des DSL, le compilateur peut transformer un programme écrit à l'aide d'un DSL en un programme équivalent dans un langage généraliste. Ce programme généraliste peut alors utiliser les opérations fournies par la couche d'abstraction précédemment décrite à la section 4.4.1. La complexité liée aux différences structurelles entre un programme écrit dans un DSL, et son programme équivalent écrit dans un langage généraliste, est ainsi réduite. La réduction du pas de compilation, grâce à l'utilisation d'une couche d'abstraction, permet de faciliter le développement d'un compilateur pour un DSL. Les programmes générés par le compilateur sont alors aussi performants que leur équivalent écrit à la main, sans le DSL, et utilisant la même couche d'abstraction.

4.5 Bilan

Le processus de développement d'un DSL, Figure 4.10, est un processus itératif qui comprend une analyse de domaine, la définition du langage et son implémentation.

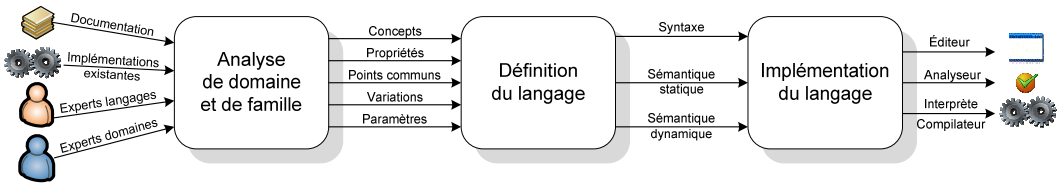


Fig. 4.10: Processus de développement d'un DSL

Pour limiter le coût du développement logiciel sur le long terme, il est possible de concevoir un DSL lorsqu'une famille de programmes doit être développée. Le DSL facilitera le développement de membres de la famille, systématisera la réutilisation de code et améliorera la fiabilité des systèmes sur lesquels les programmes s'exécuteront. Le succès d'un DSL

dépend cependant d'une bonne analyse de domaine. D'un point de vue architecture logicielle, cette analyse permet d'exprimer les éléments pour développer une couche d'abstraction (non représentée sur la figure) idéale et adaptée pour résoudre les problèmes du domaine considéré. D'un point de vue langage de programmation, les concepts et propriétés du domaine permettent de concevoir des éditeurs et de définir les analyses à réaliser sur les programmes écrits dans un DSL. Les pratiques propres au domain ainsi que les points communs, les variations et les paramètres permettent quant à eux de définir une syntaxe et une sémantique.

Les langages dédiés introduisent des constructions syntaxiques pour accéder à la couche d'abstraction. La formalisation de la sémantique du langage permet quant à elle de définir sans ambiguïté la signification des constructions du langage. Des propriétés, parfois critiques, du domaines sont alors prouvées sur les programmes grâce aux abstractions et à leur sémantique. Ces propriétés sont généralement indécidables si un langage généraliste est utilisé. De plus, ces analyses permettent d'automatiser des optimisations dans le code généré indépendamment de l'expertise du développeur.

Diverses analyses pouvant être réalisées statiquement sur les programmes avant leur exécution, les programmes développés avec une approche DSL sont plus fiables que leur homologues écrits à la main. Il est également possible de réaliser systématiquement des vérifications dynamiques dans le code généré lorsque les erreurs proviennent de données qui ne seront connues qu'à l'exécution.

Chapitre 5

Solutions existantes

Le protocole SIP présente l'avantage d'être un standard ouvert pour réaliser des communications sur un réseau IP. Il se concentre sur la signalisation en permettant d'émettre des messages, de souscrire et recevoir des événements et d'établir des sessions. Il délègue au protocole RTP le transport des flux multimédia. SIP est un protocole textuel facilement extensible. De nombreuses implémentations pour le support protocolaire de SIP sont disponibles. Dans ce chapitre, nous présentons les différentes solutions existantes pour le développement de services de téléphonie IP fondés sur SIP. Nous étudions d'une part les approches fondées sur des langages généralistes, et d'autre part les approches fondées sur des langages dédiés.

Lorsque cela sera pertinent, nous illustrerons les approches présentées par un service de redirection sur non réponse. Ce service consiste à rediriger les appels n'ayant pas abouti au propriétaire du service vers une secrétaire. Nous considérons l'utilisateur `sip:boss@example.com` comme le propriétaire du service. S'il ne répond pas ou refuse les appels qui lui sont adressés, ses appels sont redirigés vers sa secrétaire. Sa secrétaire est identifiée sur le réseau SIP par l'URI `sip:secretary@example.com`.

5.1 Langages généralistes

Les premières méthodes de développement que nous allons voir consistent à utiliser une bibliothèque de fonctions ou un intergiciel fondé sur SIP. Ces bibliothèques et intergiciels sont écrits dans différents langages de programmation généralistes et fournissent des niveaux d'abstraction variables.

5.1.1 Supports protocolaires

L'apparition d'un nouveau protocole implique le développement de code fournissant du support protocolaire, on parle communément de piles de protocole. L'objectif d'une pile est de réaliser l'analyse et la construction des messages du protocole telles que définies dans le standard. Une pile protocolaire vérifie également certaines propriétés de cohérence des messages. Il est fréquent que le niveau d'abstraction fourni reste relativement bas. Les développeurs ajoutent alors une surcouche logicielle permettant de disposer de fonctions plus abstraites. Cette surcouche, de plus haut niveau d'abstraction, traite automatiquement certaines opérations sur les messages comme la construction d'une réponse type pour une requête donnée. Il existe de fait une variété de supports protocolaires offrant divers niveaux d'abstraction et mod-

èles de programmation. Nous pouvons ainsi catégoriser les piles SIP en fonction du paradigme de programmation offert. D'un côté, les piles oSIP [Moi01], eXoSIP [Moi03], PJSIP [Pri] et sofia-sip [Nok] sont développées à l'aide du langage C et offrent donc un paradigme procédural. D'un autre côté, les piles JAIN SIP [JSR06], dissipate [Big], resiprocate [reS] ou OpenSipStack [Opeb] sont développées à l'aide de langages orienté objets, Java et C++ en l'occurence, et offrent donc un paradigme orienté objets.

5.1.1.1 Supports protocolaires procéduraux

La pile oSIP [Moi01] fournit une interface de programmation (en anglais API pour *Application Programming Interface*) permettant de manipuler les messages SIP en langage C. Le développement de tous les types de nœuds SIP, d'un client à un serveur mandataire en passant par un serveur d'enregistrement, est possible avec la pile oSIP. La pile eXoSIP [Moi03] élève les abstractions fournies par la pile oSIP, sur laquelle elle repose, pour simplifier le développement de clients SIP. Elle fournit pour cela du support pour des tâches spécifiques aux clients SIP. Parmi les opérations qui sont facilitées, on trouve par exemple l'enregistrement sécurisé auprès d'un serveur d'enregistrement, ou le maintien à jour de l'enregistrement.

L'utilisation d'une API en langage C permet d'écrire des programmes efficaces mais requiert beaucoup de compétences et de rigueur de la part du développeur. Par exemple, il doit gérer les allocations et les libérations de mémoire relatives à chaque élément d'un message SIP. Il s'agit d'un aspect critique pour du code destiné à s'exécuter sur un serveur durant de longues périodes ininterrompues. Dans ce contexte qui requiert une importante expertise, il est impossible de garantir la validité des programmes fondés sur une API C et *a fortiori* le respect des contraintes du protocole SIP. Par exemple, il n'est pas possible de garantir qu'une réponse est retournée à l'émetteur d'une requête. Il s'agit là d'une lacune structurelle propre aux langages généralistes.

5.1.1.2 Supports protocolaires orientés objets

L'API JAIN SIP [DRM04, JSR06], Java Advanced Intelligent Network SIP, permet aux développeurs Java de manipuler des messages SIP à l'aide d'un modèle à événements. L'architecture logicielle, entre une application et une pile implémentant l'API JAIN SIP, respecte le motif de code d'un écouteur *(listener pattern)*. Une application implémente une interface de l'API. Chaque méthode de cette interface sert de point de rappel à une pile. L'application souscrit auprès d'une pile afin d'être notifiée à chaque événement protocolaire. Le code utilisateur d'une application est ainsi invoqué, par exemple, à la réception de messages réseaux ou à l'expiration d'alarmes protocolaires.

L'API JAIN SIP 1.2 est constituée de 102 classes Java, totalisant plus de 500 méthodes auxquelles il faut ajouter certaines classes et méthodes spécifiques à une implémentation. Par ailleurs, l'implémentation de référence comporte plus de 300 classes totalisant près de 2 600 méthodes. Bien que la programmation orientée objet facilite le développement par rapport à une pile C, son usage n'en demeure pas moins réservé à un public averti.

L'utilisation de ces piles protocolaires oblige les développeurs à avoir une connaissance parfaite du protocole SIP. De plus, ils ont accès à l'intégralité du message sans limitation sur les opérations possibles. Il leur est ainsi possible de modifier le message, d'ajouter des en-têtes ou d'en supprimer. Si ces opérations sont nécessaires, leur utilisation doit être encadrée car elles peuvent compromettre la validité du message. Un message invalide peut corrompre une plate-

forme de communications conduisant au mieux à une exécution dégradée de la plate-forme ou au pire à son arrêt inopiné. Ce problème de vérification des programmes est inhérent à toutes piles protocolaires car elles sont fondées sur des langages généralistes. Elles doivent offrir un large panel de fonctionnalités aux développeurs mais cela se fait au détriment de la fiabilité de l'application ainsi développée. La fiabilité repose alors sur l'expertise et la bienveillance du développeur.

5.1.2 Intergiciels

Les piles protocolaires fournissent un service de base pour le support d'un protocole. Le protocole SIP est utilisé par un nombre limité de type d'entités utilisant des fonctionnalités spécifiques. On trouve dans un réseau SIP, outre les clients, des serveurs d'enregistrement, des serveurs mandataires, des serveurs de redirection et des serveurs d'applications. Les besoins de ces différents types d'entités varient. Dans un souci de simplification du développement, le niveau d'abstraction fourni par le support protocolaire a été élevé pour faciliter le développement de logiques spécifiques à une entité SIP. Les intergiciels répondent à ce besoin en fournissant un environnement de développement contrôlé pour le développement de serveurs d'applications. Nous allons voir deux paradigmes reposant sur l'API JAIN SIP pour le développement de serveurs d'applications.

5.1.2.1 SIP Servlets

Le protocole SIP partage la même structure de messages que le protocole HTTP. À l'instar des servlets HTTP [Micf], le paradigme de programmation des servlets SIP [JSR03] associe une méthode Java à chaque type de messages SIP. Nous illustrons la technologie des SIP servlets à l'aide de l'exemple introduit en tête de chapitre et qui réalise une redirection d'appels sur non-réponse (Figure 5.1, page suivante). Après les méthodes permettant de gérer le service par inversion de contrôle (lignes 6 à 13), nous trouvons la méthode `doInvite` pour la gestion des nouveaux appels. Les appels sont marqués comme nécessitant une redirection en cas d'erreur (ligne 19), puis la méthode est relayée normalement (ligne 20). Si l'appel est rejeté, la méthode `doErrorResponse` est invoquée. Si l'appel était marqué (ligne 31), une redirection a lieu (lignes 32 à 40) et le marqueur est supprimé (ligne 41). Par défaut, la réponse est relayée normalement (lignes 43 et 47).

De manière générale, à la réception d'un message SIP, la méthode Java correspondante est invoquée. La correspondance se fait sur la méthode pour une requête (par exemple, méthode SIP `INVITE` et méthode Java `doInvite` ligne 16) ou sur le code de retour pour une réponse (par exemple, code d'erreur et méthode Java `doErrorResponse` ligne 27). L'API des SIP Servlets offre des fonctions plus abstraites que celles fournies par l'API JAIN SIP. Il est ainsi possible de directement manipuler et configurer un objet mandataire, émettant et relayant des messages SIP (lignes 35, 36 et 40). Enfin, il est possible d'instancier des objets de type `SipApplicationSession` permettant de créer des sessions SIP. Ils peuvent être utilisés pour manipuler la signalisation.

Le modèle de programmation de SIP servlets est proche du protocole SIP. À chaque message SIP est associé une méthode Java. Il est adapté au développement de prototypes car il propose un modèle simple dont la mise en œuvre est rapide.

Comme l'illustre la figure 5.1, un service en SIP servlets présente de nombreux détails d'implémentation comme les appels à `super` (lignes 7, 12 et 20) qui ne sont pas spécifiques

```
1  package fr.inria;
2
3  public class RedirToSec extends SipServlet {
4
5    /** This is called by the container when starting up the service. */
6    public void init() throws ServletException {
7      super.init();
8    }
9
10   /** This is called by the container when shutting down the service. */
11   public void destroy() {
12     super.destroy();
13   }
14
15   /** This is called by the container when an INVITE message arrives. */
16   protected void doInvite(SipServletRequest request)
17     throws ServletException, IOException {
18
19     request.getSession().setAttribute("redir", Boolean.TRUE);
20     super.doInvite(request);
21   }
22
23   /**
24   * This is called by the container when an error is received
25   * regarding a sent message, including timeouts.
26   */
27   protected void doErrorResponse(SipServletResponse response)
28     throws ServletException, IOException {
29
30     SipSession session = response.getSession();
31     if((Boolean)session.getAttribute("redir") {
32       ServletContext sc = getServletContext();
33       SipFactory factory =
34          (SipFactory)sc.getAttribute("javax.servlet.sip.SipFactory");
35       Proxy p = response.getProxy(true);
36       p.setSupervised(true);
37
38       try {
39         SipURI sec = factory.createSipURI("secretary", "example.com");
40         p.proxyTo(sec);
41         session.setAttribute("redir", Boolean.FALSE);
42       } catch(Exception e) {
43         super.doErrorResponse(response);
44       }
45     }
46     else {
47       super.doErrorResponse(response);
48     }
49   }
50 }
```

FIG. 5.1: Redirection d'appels en SIP servlet

à la logique du service mais permettent l'intégration du service dans l'intergiciel des SIP servlets. Le développeur doit de plus gérer explicitement l'état pour savoir si la redirection vers la secrétaire a déjà eu lieu ou pas (lignes 19, 30, 31 et 41). L'adresse de la secrétaire est une chaîne de caractères (ligne 39) qui provoquera une erreur dynamique si elle ne représente une URI SIP valide (ligne 42). Enfin, le service s'accompagne d'une description XML de 28 lignes (non représentée) pour le déploiement du service. Cette description permet de faire le lien entre le service et le contexte d'invocation. Dans le cas présenté, le service est déclenché lorsqu'un appel, c.-à-d. une requête `INVITE`, est à destination de l'utilisateur du service.

5.1.2.2 JAIN Service Logic Execution Environment

L'intergiciel JAIN SLEE [JSR04] fournit aux développeurs un cadre de programmation plus générique que les SIP servlets. Cet intergiciel n'est pas spécifique au protocole SIP et supporte d'autres protocoles de télécommunications. Le modèle de programmation est un modèle à composants, SBB (*Service Building Block*), similaire à celui des EJB, *Entreprise Java Beans* [Mice]. L'interaction avec le réseau est réalisée grâce à des adaptateurs de ressource. Chaque adaptateur permet d'accéder à un réseau de télécommunications particulier en fournissant le support protocolaire nécessaire.

L'intergiciel JAIN SLEE offre l'avantage de fournir un modèle de programmation indépendant du protocole de communications sous-jacent. Cet intergiciel fournit également des services annexes comme la gestion d'alarmes, de la journalisation et de la gestion de services. Malheureusement, cela se traduit par un modèle de programmation beaucoup plus complexe à mettre en œuvre pour les développeurs. La documentation est quatre fois plus volumineuse que celle des SIP servlets et ne facilite pas l'accès aux développeurs occasionnels qui voudraient, par exemple, rapidement disposer d'un service de redirection. Les développeurs doivent non seulement avoir de très bonnes connaissances en télécommunications, et en SIP en particulier, mais également en programmation Java dans un environnement à composants similaire aux EJB. De plus, pour le développement de services SIP, le développeur doit manipuler les messages SIP avec l'API JAIN que nous venons de présenter. Il n'y a alors que peu d'intérêt à utiliser l'intergiciel JAIN SLEE si l'unique adaptateur de ressources actif est l'adaptateur SIP. Enfin, comme toutes les autres approches fondées sur des langages généralistes, il n'est pas possible de garantir que les programmes développés respectent des propriétés critiques de la téléphonie. Par exemple, il n'est pas possible de vérifier qu'un service JAIN SLEE traite correctement un appel, soit en renvoyant une réponse d'erreur, soit en relayant la requête.

De manière générale, il n'est pas possible de garantir la correction d'un programme, écrit dans un langage généraliste, vis à vis de propriétés spécifiques à un domaine tel que la téléphonie SIP. Pour favoriser le développement de services de téléphonie tout en préservant des propriétés critiques du domaine, des approches fondées sur les langages dédiés ont été développées.

5.2 Langages dédiés

L'alternative aux intergiciels consiste à définir un langage dédié au développement de services de communications. Nous allons à présent voir cinq langages qui ont été développés afin de faciliter l'écriture de services de téléphonie. Ils sont de deux types différents. On trouve d'un côté les langages XML comme CPL [LWS04], LESS [WS03] ou CCXML [Aub07], et d'un autre côté les langages non-XML comme MSPL [Micd] ou SER [POJK03].

5.2.1 Langages XML

L'avantage des langages XML tient au fait que de nombreux outils sont disponibles pour manipuler les programmes. Il est ainsi aisé de réaliser l'analyse syntaxique d'un fichier XML ainsi que le parcours de l'arbre abstrait généré. Les outils d'analyse syntaxique de fichier XML s'appuient sur la grammaire du fichier à analyser. Cette grammaire peut être écrite indifféremment à l'aide d'une DTD [DTD] *(Document Type Definition)* ou d'un schéma XML [XSL]. La seconde approche est toutefois à privilégier car elle permet d'exprimer des contraintes supplémentaires sur la grammaire du langage. La validation est ainsi plus précise et il y a donc moins de programmes erronés considérés comme valides, on parle de faux positifs. Une fois l'analyse syntaxique réalisée, le parcours d'un arbre abstrait permet, par exemple, l'interprétation des programmes. Ainsi, l'utilisation des technologies XML facilite le développement des outils d'un langage dédié tel qu'un éditeur, un compilateur ou un interprète. En revanche, le développement des services s'avère plus fastidieux car les programmes sont verbeux.

5.2.1.1 CPL

Peu après la standardisation de SIP, le langage CPL [LWS04] *(Call Processing Language)* a été défini pour développer des services de routage. Il est conçu avec un objectif de simplicité et de grande robustesse des services afin que tous les utilisateurs de téléphonie SIP soient en mesure de développer leurs services, éventuellement avec l'aide d'éditeurs graphiques. Bien que CPL ne soit pas explicitement dépendant de SIP, les abstractions qu'il offre sont celles de SIP (par exemple, les messages d'erreurs). Dans la pratique, CPL n'est pas utilisé en dehors du cadre des communications à base du protocole SIP.

La figure 5.2 implémente une redirection d'appels sur non réponse. Un appel entrant (ligne 4) est tout d'abord relayé de manière standard (ligne 5). Si l'interlocuteur décroche, une réponse de succès, c.-à-d. de la classe 200, est retournée et le service s'arrête. Sinon, quelque soit la cause d'échec (ligne 6), une nouvelle destination est sélectionnée (ligne 7) et l'appel est transféré (ligne 8).

Le langage CPL est défini par une grammaire DTD. Un service CPL est donc syntaxiquement valide s'il respecte la DTD du langage CPL (ligne 2). Le langage CPL offre des abstractions pour la redirection d'appels (lignes 7 et 8), l'enregistrement de messages dans un journal et le refus d'appels. Ces abstractions définissent des actions qui peuvent être appliquées sur une communication. Le langage CPL définit également des conditionnelles modifiant le routage des appels sur des critères tels que la source d'un appel, la destination d'un appel, l'heure d'un appel. Les services CPL s'exécutent sur un serveur dans le réseau [POJK03] plutôt que sur les téléphones clients. De ce fait, le langage CPL n'offre pas de fonctionnalité permettant de mettre un interlocuteur en attente par exemple. Enfin, les services CPL ne s'appliquent qu'à l'initialisation des appels. Une fois que l'appelant a reçu une réponse définitive, c'est-à-dire que son appel est un succès ou un échec, le service CPL s'arrête (lignes 5 et 8). Il n'est donc pas possible d'enregistrer la durée d'un appel par exemple.

De manière générale, le langage CPL a été conçu pour éliminer tout problème potentiel pouvant corrompre la plate-forme de téléphonie. Malheureusement, cela s'est fait au détriment de l'expressivité du langage qui ne propose pas de variable par exemple. De fait, il n'est pas possible d'écrire un service qui enregistre les domaines appelés. Il n'est également pas possible d'utiliser de l'information extérieure à l'appel comme une base de donnée par exemple. Seule une partie des informations issues d'une requête `INVITE` ou d'une de ses réponses finales sont

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE cpl PUBLIC "-//IETF//DTD RFC3880 CPL 1.0//EN" "cpl.dtd">
3  <cpl>
4    <incoming>
5      <proxy>
6        <default>
7          <location url="sip:secretary@example.com" clear="yes">
8            <proxy />
9          </location>
10       </default>
11     </proxy>
12   </incoming>
13 </cpl>
```

FIG. 5.2: Redirection d'appels en CPL

exploitables par un service CPL. L'exploitation de données issues du système d'information d'une entreprise permettrait d'avoir un routage plus fin et des services plus génériques car n'exprimant que la logique. L'adresse de la secrétaire (ligne 7) ne serait alors plus forcément une constante du service.

5.2.1.2 LESS

Face aux limitations du langage CPL concernant la création d'appels ou la gestion de la présence, le langage LESS [WS03], *Language for End-System Services*, a été conçu. Le langage LESS hérite du langage CPL sa représentation XML, ainsi que la plupart des concepts. Il s'adresse également au même public d'utilisateurs non-experts et non-programmeurs. Mais contrairement à CPL, le langage LESS est prévu pour s'exécuter sur un terminal. Ce positionnement ouvre un nouveau spectre de services comme nous l'avons vu dans la section 3.2.3.1. Le langage LESS est enrichi avec des concepts permettant de contrôler le terminal. Un service LESS peut ainsi créer et accepter des appels, transférer un appel en cours, créer une conférence téléphonique, envoyer des messages instantanés ou encore afficher une fenêtre graphique sur le poste client. Enfin, un schéma XML sert à la validation des services. La validation est ainsi plus précise qu'en CPL et il y a donc moins de programmes erronés considérés comme valides.

La figure 5.3 présente le service de redirection en cas d'erreur vers la secrétaire. Le service s'exécutant sur le client, une partie des erreurs gérables en CPL (par exemple, client non présent, erreur réseau) ne le sont plus dans ce contexte. L'erreur se limite donc à une non-réponse modélisée par le status `busy`. L'appel entrant est traité par la ligne 2. Si l'interlocuteur n'est pas disponible (ligne 3 et 4), une nouvelle destination est sélectionnée (ligne 5) et l'appel est transféré (ligne 6).

Le langage LESS a cependant la même limitation que le langage CPL. Il n'est pas possible de définir des variables utilisateurs et des interactions avec des systèmes extérieurs. Un autre inconvénient du langage LESS est qu'il est conçu pour fonctionner sur le poste de l'utilisateur. Comme nous l'avons vu à la section 3.2.3.1, l'emplacement de prédilection pour les services reste le réseau. Le comportement du système, vis à vis de l'utilisateur, est ainsi plus prévisible et l'administration plus simple.

```
1  <less>
2    <incoming>
3      <status-switch status-name="activity">
4        <status is="busy">
5          <location url="sip:secretary@example.com">
6            <redirect/>
7          </location>
8        </status>
9      </status-switch>
10   </incoming>
11 </less>
```

FIG. 5.3: Redirection d'appels en LESS

5.2.1.3 CCXML

Pour dépasser les limitations du langage CPL, le consortium W3C est en train de définir le langage CCXML [Aub07], également fondé sur une représentation XML. Les programmes CCXML sont validés syntaxiquement grâce à un schéma XML. La figure 5.4 implémente le service de redirection. Un appel entrant déclenche l'évaluation du bloc lié à l'événement `connection.alerting` pour l'état courant du service, c.-à-d. `init` (ligne 6). L'appel est transmis sur le téléphone de Bob (ligne 7). Si Bob décroche, le service s'arrête. Sinon la redirection échoue et les lignes 10 à 13 sont évaluées. Une redirection est alors réalisée vers la secrétaire (ligne 11) et l'état du service prend la valeur `redir` (ligne 12) afin que ce bloc de code ne soit pas réévalué en cas d'échec de la redirection vers la secrétaire.

Le modèle de programmation du langage CCXML est un automate à état, géré de manière explicite par les développeurs. Les lignes 3 et 12 illustrent la gestion de l'état par un développeur. La variable utilisateur `state0`, déclarée à la ligne 3, permet de modéliser l'état de l'automate. Des événements (lignes 6 et 11) permettent de faire évoluer l'état du service tout en effectuant des actions (lignes 7, 12 et 13). Un événement est émis vers un service à chaque modification de l'état d'une communication, par exemple un appel entrant, ou au contraire la fin d'un appel. Les développeurs définissent des fragments de code qui seront exécutés lorsque les différents événements surviendront. Le langage CCXML offre des fonctionnalités permettant de gérer des conférences afin de connecter ensemble plusieurs interlocuteurs.

L'utilisation d'une représentation XML facilite le traitement effectué au niveau du serveur car il est possible de réutiliser des outils génériques pour la manipulation du XML. Malheureusement, cela rend également le développement de services plus difficile. En effet, il est moins facile pour un programmeur de percevoir et de vérifier la logique de son application lorsque celle-ci est noyée dans l'enchevêtrement des balises XML. Il existe bien quelques éditeurs graphiques mais, une représentation graphique est peu adaptée pour représenter la logique de services complexes. Enfin, comme l'illustre l'exemple de la figure 5.4, l'état du service est explicitement modifié par le développeur pour faire évoluer l'automate sous-jacent. Cette gestion explicite, en plus d'être une charge conséquente pour le développeur, est source d'erreurs dans les services. En effet, il est facile de commettre une erreur typographique modifiant le comportement de l'automate sans qu'aucune alerte ne vienne éveiller les soupçons du développeur.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <ccxml version="1.0" xmlns="http://www.w3.org/2002/09/ccxml">
3    <var name="state0" expr="'init'"/>
4
5    <eventprocessor statevariable="state0">
6      <transition state="init" event="connection.alerting">
7        <redirect dest="sip:bob.phone@example.com"/>
8      </transition>
9
10     <transition state="init" event=connection.redirect.failed>
11       <redirect  dest="sip:secretary@example.com"/>
12       <assign name="state0" expr="'redir'" />
13     </transition>
14   </eventprocessor>
15 </ccxml>
```

FIG. 5.4: Redirection d'appels en CCXML

5.2.2 Langages non-XML

Les langages non-XML nécessitent un effort plus conséquent lors du développement d'outils pour le langage. La grammaire du langage doit, par exemple, être annotée avec des attributs. Ces attributs sont liés à un langage de programmation. La spécification de la grammaire est alors exprimée sous une forme dépendante d'un langage d'implémentation comme C pour Yacc [Joh75] ou les outils XT[BKVV08] pour Stratego [Kal06]. En contrepartie de l'effort initial, le développement de services est par la suite beaucoup plus simple et les services sont plus aisés à maintenir que leurs équivalents XML. Nous présentons ici le langage de filtrage MSPL [Micd] et le langage de configuration d'un serveur SER [POJK03].

5.2.2.1 MSPL

Microsoft dispose dans son offre logicielle de serveurs de communications implémentant un système de téléphonie SIP, Response Point [Micc] et Office Communications Server [Mica]. Afin de faciliter le développement de services de téléphonie, le langage de filtrage MSPL [Micd] est fourni. Lorsqu'une opération n'est pas disponible via le langage MSPL, la commande MSPL `Dispatch` (Figure 5.5, ligne 30) permet de faire appel à du code écrit en C# (lignes 41 à 48) et ainsi d'accéder au cadre de programmation .NET [Micb].

Le langage MSPL est cependant très restreint dans ses objectifs. Il n'est ainsi pas possible d'écrire le service de redirection vers la secrétaire entièrement en MSPL comme l'illustre la figure 5.5. En effet, un service MSPL ne permet pas de modifier et émettre une requête pendant le traitement d'une réponse. Il n'est ainsi pas possible de rediriger la requête initiale vers la secrétaire lorsque le service traite la première réponse finale et que celle-ci est un échec. De plus, il n'est pas possible pour un développeur de préserver un état entre deux invocations d'un service car le langage MSPL ne permet d'avoir que des variables locales au script. Ces variables sont donc volatiles et leur valeur est perdue chaque fois que le service est interrompu.

Le service de redirection en MSPL commence par filtrer les requêtes `INVITE` (ligne 5 et 13). La requête est marquée (ligne 17) puis transmise à tous les terminaux où le destinataire est enregistré (lignes 18 à 22). En cas d'échec (lignes 25 à 28) et si la redirection n'a pas encore eu lieu (ligne 29), le traitement de la redirection est délégué au code C# (lignes 41 à 48). Après avoir configuré une redirection (lignes 42 à 44), et modifié le marquage de la requête (ligne 45)

afin que la redirection n'ait lieu qu'une fois, la nouvelle destination est sélectionnée (ligne 46) et la requête émise (ligne 47).

```
1  <?xml version="1.0">
2  <lc:applicationManifest
3   lc:appUri="http://www.my_domain_here.com/DefaultRoutingScript"
4   xmlns:lc="http://schemas.microsoft.com/lcs/2006/05">
5  <lc:requestFilter methodNames="INVITE"
6                    strictRoute="false"
7                    registrarGenerated="true"
8                    domainSupported="true"/ >
9  <lc:responseFilter reasonCodes="ALL" />
10 <lc:proxyByDefault action="true" />
11 <lc:splScript><![CDATA[
12
13 if(sipRequest) {
14   toUri = GetUri( sipRequest.RequestUri);
15   toUserAtHost = Concatenate( GetUserName( toUri ), "@", GetHostName( toUri ) );
16
17   sipRequest.Stamp = "normal";
18   BeginFork(false,0);
19   foreach (dbEndpoint in QueryEndpoints( toUserAtHost)) {
20     Fork(dbEndpoint.ContactInfo);
21   }
22   EndFork();
23 }
24
25 if(sipResponse) {
26
27   if (sipResponse.StatusClass != StatusClass._1xx) {
28     if (sipResponse.StatusClass != StatusClass._2xx) {
29       if (sipResponse.Stamp == "normal") {
30         Dispatch("toSecretary");
31         return;
32       }
33     }
34   }
35   ProxyResponse();
36 }
37
38 ]]></lc:splScript>
39 </lc:applicationManifest>
40 ...
41 public void toSecretary (object sender, ResponseReceivedEventArgs responseArgs) {
42   ServerTransaction st = responseArgs.ClientTransaction.ServerTransaction;
43   ClientTransaction ct = st.CreateBranch();
44   Request req = st.Request.Clone();
45   req.stamp = "secretary";
46   req.RequestUri = "sip:secretary@example.com";
47   ct.sendRequest(req);
48 }
```

FIG. 5.5: Redirection d'appels en MSPL

5.2.2.2 SER et OpenSER

Les serveurs SER [POJK03] et OpenSER [Opea] sont des serveurs mandataires SIP. Le serveur OpenSER est historiquement fondé sur SER. SER est réputé pour sa stabilité. OpenSER quant à lui privilégie l'innovation et propose de nombreuses extensions, parfois au

détriment de la fiabilité. Grâce à l'ajout de modules écrits en C, il est possible d'étendre les fonctionnalités offertes par ces serveurs. Les modules exportent des fonctions. Lorsqu'un module est chargé à la demande du fichier de configuration (Figure 5.6, lignes 4 et 5), les fonctions qu'il exporte, peuvent alors être invoquées par l'administrateur du serveur dans le reste du fichier de configuration. Cette architecture modulaire permet par exemple d'exploiter de manière uniforme plusieurs implémentations de base de données, de gérer la présence, de fournir l'interopérabilité avec les réseaux XMPP. Il existe également un module CPL permettant aux utilisateurs de déployer leurs services.

```
1  ...
2  mpath="/usr/local/lib/openser/modules"
3  ...
4  loadmodule "registrar.so"
5  loadmodule "sl.so"
6  loadmodule "tm.so"
7  ...
8
9  route {
10   ...
11   if(isMethod("INVITE")) {
12     lookup("location");
13     if (search("(t|To):boss@example.com")) {
14       t_on_failure(1);
15     }
16     if(!t_relay()) {
17       sl_send_reply("500", "relaying failed");
18       break;
19     }
20   }
21   ...
22 }
23
24 failure_route[1] {
25   append_branch();
26   rewriteuri("sip:secretary@example.com");
27   lookup("location");
28   if(!t_relay()) {
29     sl_send_reply("500", "relaying failed");
30   }
31 }
```

FIG. 5.6: Redirection d'appels en OpenSER

La figure 5.6 illustre des extraits d'un fichier de configuration d'un serveur OpenSER qui réalisent le service de redirection vers la secrétaire. Le fichier se décompose en deux parties : d'une part un préambule permet la configuration du serveur et des modules, d'autre part un ensemble de blocs de code exprimant la logique de routage des messages SIP. Toutes les requêtes SIP reçues par le serveur sont traitées en premier lieu par le bloc `route` (lignes 9 à 22). Lorsque la méthode de la requête est un `INVITE` (ligne 11), le serveur d'enregistrement est consulté (ligne 12). Dans le cas où le destinataire est concerné par le service (ligne 13), la transaction SIP est annotée (ligne 14) afin que le bloc de redirection en cas d'erreur (lignes 24 à 31) soit ultérieurement invoqué si nécessaire. Le traitement de la requête `INVITE` se termine ensuite normalement et celle-ci est transmise (ligne 16) au contact fourni par le serveur d'enregistrement. Si l'appel échoue, le bloc `failure_route` précédemment sélectionné est exécuté.

Après avoir initialisé la redirection (ligne 25), la nouvelle destination est définie (ligne 26), et l'appel émis (ligne 28).

Ce mode de développpement de services présente de nombreux inconvénients. Tout d'abord, une très grande expertise est nécessaire tant sur SIP que sur l'implémentation du serveur et son API. Ensuite, l'ensemble des services est fusionné dans un seul et unique fichier dont seul l'administrateur du serveur possède les droits d'accès. Enfin, aucune garantie n'est fournie au développeur concernant la validité de la logique de son application. Finalement, lorsqu'aucun module ne permet de résoudre un problème, il est possible de développer son propre module d'extension. Toutefois, cette solution présente, en plus des mêmes défauts, le besoin d'une expertise en programmation bas-niveau C et une connaissance encore plus grande de l'implémentation du serveur.

5.3 Bilan

Il existe deux types de solutions au développement de services de communications fondés sur le protocole SIP. D'une part, les approches généralistes permettent de programmer des services arbitraires grâce aux langages généralistes et à de vastes API. Cependant, ces approches ne garantissent pas la fiabilité des services. La correction des programmes est entièrement à la charge des développeurs. Ceux-ci doivent disposer d'une grande expertise couvrant plusieurs domaines comme les télécommunications, les systèmes distribués et la programmation informatique. D'autre part, les approches langages dédiés actuelles n'offrent pas d'alternative conciliant fiabilité des services, expressivité et concision. En effet, le langage MSPL n'est prévu que pour le filtrage de messages et délègue la logique des services à l'API généraliste du serveur. Les développeurs font alors face à l'écueil des approches généralistes. Le langage CPL offre un niveau de garantie intéressant mais au détriment de l'expressivité des services qui ne tirent pas parti des ressources du réseau. Le langage LESS est quant à lui destiné aux terminaux et non pas aux serveurs d'applications. Le langage CCXML fournit une bonne expressivité mais pas de vérification autre que syntaxique. De plus, il est particulièrement verbeux et offre un modèle de programmation propice aux erreurs. Enfin, les scripts de configuration des serveurs SER ou OpenSER ne permettent pas un développement et un déploiement simple des services car l'intervention de l'administrateur du serveur est requise et le redémarrage du serveur nécessaire.

Par rapport à la dichotomie des types de services (les services de routage et les services entités), les approches généralistes sont plus souples et permettent le développement des deux types de services. Elles n'offrent cependant aucun support pour gérer la diversité des types de données et peu ou pas de support pour gérer les modes de communications. Quant aux approches dédiées, les possibilités sont variables. Les serveurs SER et OpenSER ne permettent de réaliser que du routage. Le langage CPL est également réservé aux services de routage tandis que le langage CCXML permet de développer des services entités, notamment grâce à ces fonctionnalités de conférence. Le langage LESS permet quant à lui de développer des services entités mais uniquement sur les terminaux, ce qui limite les possibilités offertes. Finalement, aucune approche dédiée ne prend en compte la diversité des types de données. D'un point de vue génie logiciel, aucune approche dédiée ne fournit de sémantique formelle du langage qu'elle propose. Les subtilités d'implémentation sont ainsi laissées à la discrétion des développeurs et à leur compréhension de la documentation en langage naturel, parfois soumise à interprétation.

Chapitre 6

Démarche suivie

Dans le chapitre 2, nous avons caractérisé les communications informatiques et mis en évidence le besoin de services pour les gérer. Nous avons également vu l'existence de deux types de services, d'une part les services de routage et d'autre part les services entités. La téléphonie SIP, présentée dans le chapitre 3, fournit un cadre ouvert englobant l'ensemble des modes de communications pour divers types de données. Ce cadre ouvert et programmable permet le développement de services de communications. De plus, il peut être étendu aisément à de nouveaux types de données facilitant l'intégration de futurs entités communicantes. Les approches existantes pour le développement de services de communications, présentées au chapitre 5, n'offrent cependant aucune solution satisfaisante. Aucune n'est à la fois simple, sûre et expressive. L'approche des langages dédiés a prouvé son efficacité sur ces points dans de nombreux domaines comme le montrent les exemples présentés au chapitre 4.

Nous récapitulons, dans ce chapitre, les problèmes auxquels doit faire face un développeur de services de communications. Nous présentons ensuite l'approche langage, fondée sur le concept des langages dédiés que nous préconisons d'adopter.

6.1 Problématique

Nous avons vu que les communications ne cessaient de se multiplier (Chapitre 2). De nouvelles entités communicantes apparaissent ainsi régulièrement. Elles définissent parfois de nouveaux types de données. Le protocole SIP, initialement conçu pour la téléphonie IP, permet plusieurs modes de communications : les communications synchrones, qu'elles soient sporadiques ou par flux de données, et les communications asynchrones. De plus, le protocole SIP est extensible par nature. Il est donc parfaitement adapté comme support des communications et de leur évolution. Afin de faire face à l'augmentation des communications et de faciliter leur gestion, il est nécessaire d'automatiser le traitement grâce au développement de services de communications.

L'importance des communications impose cependant que la fiabilité des services de communications soit garantie avant leur exécution. La plate-forme de communications est ainsi protégée contre les erreurs que la logique des services applicatifs peut introduire. Toutefois, il n'existe aucune solution satisfaisante pour le développement de services de communications. D'un côté, les solutions généralistes sont complexes à mettre en œuvre. De plus, elles ne permettent pas de garantir des propriétés critiques de la plate-forme de communications sous-jacente. D'un autre côté, les solutions dédiées existantes n'offrent pas de compromis sat-

isfaisant entre les vérifications possibles, l'expressivité et la simplicité de développement. Il est ainsi nécessaire de choisir, par exemple, entre CPL pour la simplicité et les vérifications, et CCXML pour l'expressivité.

Le besoin urgent de services, ainsi que les conséquences que peut avoir le déploiement d'un service erroné, justifient le développement d'une nouvelle approche à la fois simple, expressive et sûre. Nous préconisons la mise en œuvre d'une approche fondée sur le concept des langages dédiés. En effet, l'expérience montre que grâce aux restrictions et aux abstractions qu'ils proposent, les langages dédiés permettent de spécifier des solutions à une famille de problèmes, tout en rendant décidable des propriétés spécifiques au domaine pour lequel ils sont dédiés. Nous présentons maintenant la démarche suivie pour la conception et la réalisation de cette nouvelle approche dans le cadre des services de communications.

6.2 Démarche globale

L'approche que nous proposons pour faciliter le développement de services de communications est fondée sur l'utilisation des langages dédiés, tels que présentés dans le chapitre 4. Comme nous l'avons vu (Section 2.2), il existe deux types de services : d'une part les services de routage, et d'autre part les services entités. La première étape de notre démarche a donc consisté à définir un langage dédié au développement des services de routage. Par la suite, nous avons généralisé le développement de services aux services entités en définissant un second langage. Chacun de ces deux langages a fait l'objet d'une étude de domaine avant sa conception. Des sémantiques statiques des langages ont été définies et permettent différentes analyses en fonction du type de services. Enfin, une sémantique dynamique pour chacun de ces deux langages en définit le comportement à l'exécution. L'ensemble des sémantiques a été utilisé pour le développement d'une implémentation de chacun des langages.

6.2.1 Un langage pour les services de routage

La figure 6.1 illustre le processus de développement des services de routage. Un service est tout d'abord analysé afin de déceler d'éventuelles erreurs ou incohérences qui pourraient compromettre la plate-forme SIP. Cette phase d'analyse détecte, par exemple, les services qui ne retournent pas de réponse à la suite d'une requête ou les services susceptibles de violer une règle du protocole SIP. L'objectif est de rejeter lors de l'analyse les services qui risquent de placer la plate-forme de communications dans un état incohérent. Enfin, si le service est correct, celui-ci est déployé sur le serveur d'applications après une phase de transformation. Il est alors possible lors de cette transformation de réaliser automatiquement certaines optimisations. Ces optimisations peuvent, par exemple, permettre de minimiser l'espace mémoire nécessaire pour un service donné ou d'agir sur des paramètres du protocole SIP afin de faciliter le passage à l'échelle du serveur d'applications.

6.2.2 Un langage pour les services entités

La figure 6.2 illustre le développement des services entités. Un expert modélise un environnement comme par exemple une entreprise, une maison, une administration ou un hôpital. Selon l'environnement, divers scénarios peuvent être envisagés, surveillance d'un patient dans

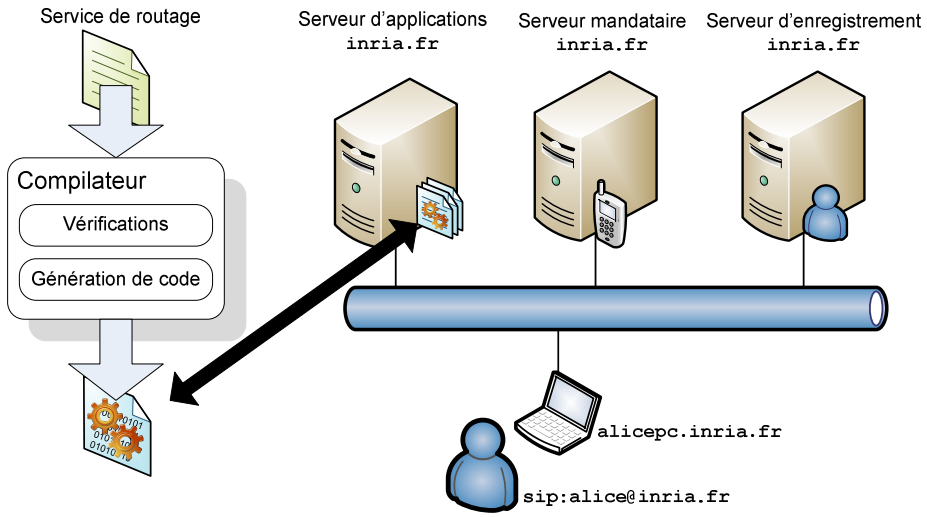


FIG. 6.1: Processus de développement de services de routage

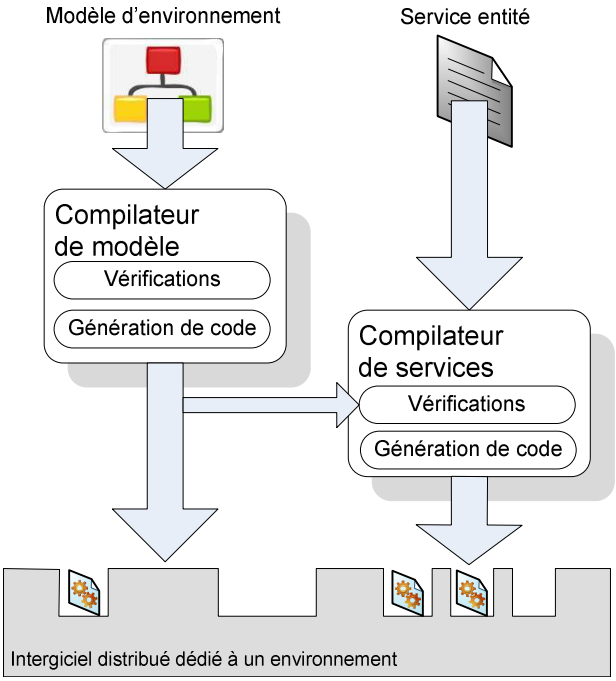


FIG. 6.2: Processus de développement de services entités

un hôpital, localisation de biens ou de personnes dans un bâtiment, redirection d'appels « intelligente » dans une administration. La modélisation d'un environnement consiste à définir les services présents et nécessaires pour des scénarios d'applications. Un service abstrait définit une classe de services entités similaires. Chaque définition abstraite indique les fonctionnalités qui permettent de communiquer et les propriétés qui caractérisent une classe de services. L'ensemble des services abstraits constitue le modèle d'un environnement. Un modèle d'environnement, ou par abus de langage un environnement, spécifie également les communications entre les services abstraits et les types de données mis en œuvre par ses communications. Il est ainsi possible de définir une variété d'environnements où des services distribués communiquent entre eux. Nous nous intéresserons plus particulièrement à la définition d'un environnement pour la téléphonie incluant, par exemple, des annuaires, des boîtes vocales, des bases de données et des téléphones. Une fois un environnement modélisé, un ou plusieurs services entités sont développés pour des services abstraits. Ces services entités spécifient des logiques de coordination des communications entre les services d'un environnement. Ces logiques sont exprimées à l'aide d'un langage dédié qui réutilise les définitions d'un environnement et partage des concepts liés au domaine avec le langage de modélisation.

Après avoir vérifié la cohérence d'un environnement et des communications entre ses services, un premier compilateur génère un intergiciel dédié à la spécification d'un environnement donné. Cet intergiciel offre du support de programmation spécifique aux services et aux types de données définis dans l'environnement. Ce support facilite le développement manuel, dans un langage généraliste, de services communicants de l'environnement. Un second compilateur permet de générer des services pour cet intergiciel à partir de services entités écrit à l'aide du langage de logique de coordination que nous proposons. Le second compilateur permet de vérifier les services entités vis à vis du service abstrait qu'ils implémentent. Ils doivent notamment respecter les contraintes du service en terme de fonctionnalités et de propriétés tel que définies par le modèle de l'environnement.

Dans la suite de ce document, nous détaillons les différentes étapes de la démarche que nous venons de présenter. Tout d'abord, nous décrivons la conception d'un langage dédié pour le développement de services de routage. Nous présentons l'analyse de domaine, qui reprend les éléments principaux déjà abordés dans l'ensemble de cette partie, puis l'analyse de propriétés du domaine, et enfin l'implémentation et la mise en œuvre de la solution. De manière similaire, nous présentons ensuite la généralisation qui consiste à définir un langage pour le développement de services entités.

Deuxième partie

Approche proposée

Chapitre 7

SPL

[1]

Notre objectif est de concevoir un langage dédié au développement de services de routage. La conception du langage dédié SPL *(Session Processing Language)* suit la variante de la méthodologie Sprint [CM98] présentée au chapitre 4 et fondée sur l'utilisation d'une sémantique opérationnelle et d'une machine virtuelle. Après une étude du domaine des services de routage dans les systèmes de communications SIP, nous définissons le langage SPL. La définition de ce langage comprend, outre la syntaxe, des analyses fondées sur une sémantique statique ainsi qu'une sémantique dynamique décrivant le comportement des programmes. Cette sémantique dynamique sert de fondation à l'implémentation d'interprètes pour le langage SPL.

7.1 Analyse du domaine

La conception d'un DSL nécessite une étape préliminaire et cruciale : l'étude du domaine qu'il cible. Nous allons dans cette section récapituler les éléments spécifiques aux services de routage fondés sur le protocole SIP. Ces éléments définissent le domaine du futur langage. Ils sont abordés au travers d'une étude qui présente les sources d'information, les points communs et les variations du domaine. Cette étude se termine par un cahier des charges pour la conception du langage.

7.1.1 Sources d'information

Afin de mener à bien l'analyse du domaine, diverses sources d'information ont été utilisées. Nous avons notamment recensé les services de routage existants dans les solutions patrimoniales. Nous avons ensuite étudié plus en détail l'ensemble des approches de développement logiciel reposant sur le protocole SIP ainsi que les services fondés sur ces approches (Chapitre 5). L'analyse de ces éléments nous a permis de déterminer la terminologie, les points communs et les variations du domaine. Enfin, l'étude approfondie des différentes RFC [Nie04, RSC+02,

[1] Illustration de Pessin tirée de l'article sur SPL dans l'*Inédit* n°51, novembre 2005, INRIA

Roa02, CRS+02, SIM] spécifiant le protocole SIP et ses extensions a permis de définir les contraintes imposées par le protocole et devant être respectées par les services.

7.1.2 Points communs

L'objectif de l'analyse de points communs est de déterminer les concepts relatifs au domaine étudié. Ces concepts seront présents dans l'ensemble des programmes sous la forme de mots-clés ou de constructions du langage. Nous pouvons distinguer deux types de concepts : les objets et les opérations. Un objet est utilisé pour modéliser des données du domaine, comme un message SIP dans notre cas. Les opérations définissent les manipulations possibles sur un objet, comme le routage d'un message par exemple.

7.1.2.1 Objets

Les objets représentent les concepts du domaine qui sont explicités. Les messages constituent les objets de base de notre domaine. Ils peuvent être soit des requêtes, soit des réponses. Les messages sont composés d'en-têtes suivis d'une charge utile. Les en-têtes sont riches d'informations renseignées par les nœuds SIP et constituent des objets qu'il est important de pouvoir manipuler. Enfin, SIP est un protocole textuel inspiré du protocole HTTP qui repose sur un usage important des URI. En effet, le routage des messages SIP consiste à manipuler et réécrire des URI. Les URI constituent donc des objets qui doivent pouvoir être manipulés de façon sûre par un développeur afin de ne pas compromettre le routage.

Événements Dans les approches existantes, tous les événements protocolaires pris en compte, sont modélisés par un événement au niveau programmation. La réception de messages (des requêtes ou des réponses), ainsi que l'expiration d'alarmes protocolaires, provoquent l'exécution de la logique correspondante, définie par l'utilisateur.

Dans le cas de JAIN SIP par exemple, le développeur est notifié de trois types d'événements : la réception de requêtes, la réception de réponses et l'expiration d'alarmes. Les événements des SIP servlets sont plus précis et il est ainsi possible d'avoir un événement pour chaque type de requête (par exemple, `INVITE`, `ACK`, `BYE`). Toutefois, ce niveau d'abstraction manque encore de finesse et de précision. En effet, une requête `INVITE` peut être émise à la fois pour créer une communication mais également pour modifier les propriétés d'une communication existante (par exemple, pour ajouter de la vidéo). De même, une requête `REGISTER` sert à la fois à l'enregistrement initial auprès du réseau SIP mais également au maintien d'un enregistrement existant. Enfin, le désenregistrement est également réalisé via une requête `REGISTER`. Nous proposons donc de raffiner les événements protocolaires en fonction de leur signification. Lorsque qu'il s'agit de la première transmission, l'événement porte le même nom que la requête. Lors des transmissions suivantes, sa signification diffère et il est préfixé par `RE` ; on parle d'événements raffinés. Nous avons ainsi les événements `REREGISTER` et `REINVITE` qui raffinent, respectivement, les événements protocolaires de réception de requêtes `REGISTER` et `INVITE`.

Lorsque l'on considère les approches généralistes, celles-ci exposent également des événements de la plate-forme, comme les méthodes `init` et `destroy` pour les SIP servlets (Figure 5.1, page 48), permettant à l'utilisateur d'initialiser ou détruire un état relatif au service. Cet état peut aller d'une simple variable, comme un compteur, à une référence sur une base de données ou un annuaire pour réaliser des interrogations ultérieures. En plus des événements

plates-formes pour l'administration des services, nous proposons de modéliser les événements protocolaires de terminaison des communications par des événements plates-formes. Ces événements sont déclenchés systématiquement à la fin d'une communication, que celle-ci soit dûe à l'expiration d'une alarme ou à une terminaison normale à l'aide d'une requête SIP. Nous introduisons ainsi les événements `uninvite` et `unregister` marquant respectivement la fin d'une conversation et la fin d'un enregistrement.

Partitionnement des événements en sessions Que l'on considère les concepts de service, enregistrement ou dialogue, c.-à-d. une conversation, une notion de cycle de vie est présente. Nous utiliserons par la suite le terme de *session* pour désigner un cycle de vie. Il est à noter que nous avons généralisé le terme de session qui, dans la RFC définissant SIP, ne fait référence qu'à une session dialogue. Une session commence par une phase d'initialisation qui crée une instance de session particulière. À la suite de cette création, une phase intermédiaire permet la modification de la session. Enfin, la session se termine par un événement final. Les événements peuvent ainsi être partitionnés en trois ensembles selon leur rôle : création, modification et terminaison. Le tableau 7.1 montre le partitionnement des événements raffinés selon leur rôle vis à vis de la session à laquelle ils se rapportent.

Nous définissons quatre types de sessions : `service`, `registration`, `dialog` et `event`. Les sessions `service` englobent l'intégralité du cycle de vie d'un service de son activation à son arrêt. De manière similaire, les sessions `registration` regroupent les opérations ayant lieu entre le premier enregistrement de l'utilisateur sur le réseau SIP et son désenregistrement. Les sessions `dialog` et `event` correspondent respectivement aux sessions pour les communications par flux de données et aux communications par événements. Une session `event` s'active lorsqu'une entité SIP souscrit à un événement. Elle comprend les notifications des événements qui s'en suivent jusqu'à l'arrêt de la souscription.

Les requêtes `MESSAGE`, `PUBLISH` et `OPTIONS` font l'objet d'un traitement particulier. La requête `MESSAGE` est utilisée pour les communications synchrones sans flux de données et permet l'échange sporadique de données entre deux entités. La requête `PUBLISH` permet l'émission d'événements SIP. Cependant, cette requête n'appartient pas à une session événement `event` et est transformée en requête `NOTIFY` par le serveur de notification auquel elle est adressée. La requête `OPTIONS` permet, quant à elle, de découvrir les fonctionnalités offertes par un client ou un serveur SIP. Ces trois méthodes font l'objet d'un traitement particulier car il n'y a pas de session associée.

En-têtes La première ligne des messages SIP est suivie d'en-têtes puis d'une charge utile. Les en-têtes contiennent des informations qui doivent être accessibles par un service. Par exemple, un service de filtrage d'appels nécessite de pouvoir déterminer l'origine d'un appel, sa destination initiale ou si une redirection a eu lieu. Les messages SIP sont des messages textuels et chaque en-tête est donc représenté par une chaîne de caractères. Il est toutefois plus pratique pour un développeur d'avoir une vue structurée des en-têtes. L'en-tête `Max-Forwards`, qui évite les boucles infinies entre serveurs mandataires, peut ainsi être vue comme un entier non signé sur 8 bits, par exemple.

Uniform Resource Identifier (URI) Les URI représentent un concept clé dans les protocoles informatiques tels que SMTP ou HTTP. Le protocole SIP réutilise également ce concept.

Concepts	Événements		
	Création	Modification	Terminaison
service	deploy	refresh	undeploy
registration	REGISTER	REREGISTER	unregister
dialog	INVITE	CANCEL ACK REINVITE	BYE uninvite
event	SUBSCRIBE	RESUBSCRIBE NOTIFY	unsubscribe
Sans session	MESSAGE PUBLISH OPTIONS		

TAB. 7.1: Classification des événements SIP raffinés

Le routage des messages SIP est ainsi fondé sur la réécriture d'URI. Il est donc important de pouvoir garantir que les URI présentes dans un service sont bien formées et respectent la RFC 3986[BLFM05].

Bien que le protocole SIP manipule essentiellement des URI SIP, il est possible d'utiliser également des URI de courriers électroniques ou des URI Web. Par exemple, si le destinataire d'un appel ne peut répondre, il peut rediriger son interlocuteur vers une page Web ou sa boîte de courriers électroniques. Un service de routage SIP doit donc être capable de gérer des URI arbitraires.

État Afin de personnaliser le routage de messages SIP, les services doivent s'appuyer sur l'état du système pour prendre des décisions de routage. L'état du système est défini par l'état des communications et l'état du système d'information. Il n'est cependant pas nécessaire de définir le routage en fonction de l'état complet du système et les développeurs peuvent définir un modèle de cet état au sein d'un service grâce à des variables. Ces variables constituent un état associé au service, que l'environnement d'exécution doit manipuler automatiquement afin de faciliter la tâche des développeurs.

7.1.2.2 Opérations

L'opération fondamentale pour les services de routage consiste à router les messages. Il est toutefois nécessaire, pour rendre le langage expressif, de compléter les opérations de signalisation par des opérations plus standards, de calculs arithmétiques par exemple.

Opérations de signalisation Toutes les approches proposent des opérations permettant de relayer les messages. Nous pouvons toutefois distinguer deux opérations élémentaires pour le routage : la première consiste à relayer une requête vers le destinataire, tandis que la seconde sert à retourner une réponse vers l'initiateur de la communication.

L'opération de routage est primordiale, il est toutefois nécessaire de disposer d'autres opérations afin de manipuler les messages SIP. L'une des principales caractéristiques du protocole SIP est d'être inspiré du protocole HTTP. Un message SIP possède, en plus de la première ligne dédiée au routage, des en-têtes. Il nécessaire que le langage dispose d'opérateurs permettant de manipuler ces en-têtes.

Opérations standards Afin de rendre expressif le langage, il est important qu'il comporte des opérations standards, non dédiées à la signalisation. Ces opérations doivent permettre de réaliser, par exemple, des calculs arithmétiques et logiques. Afin de tirer parti du système d'information, le langage doit également permettre d'invoquer des méthodes externes au service et au langage.

7.1.3 Variations

Lors d'une analyse de domaine, il faut étudier les points communs, mais également les variations. Nous venons de voir les points communs des services de routage. Nous allons à présent voir les variations qui différencient ces services, d'une part celles concernant les objets précédemment décrits, et d'autre part celles concernant les opérations.

7.1.3.1 Objets

Pour chacun des objets communs aux services de routage, nous pouvons définir une dimension selon laquelle l'objet varie. Nous allons voir quelles sont les variations qui interviennent sur les événements, les sessions, les en-têtes et l'état des services.

Événements Tous les événements ne nécessitent pas de traitement particulier pour un service donné. Par exemple, les services CPL permettent uniquement le routage d'appels et définissent donc une logique de routage uniquement pour les requêtes `INVITE`. Les services équivalents dans notre langage se doivent d'être aussi simples. Les événements nécessaires à un service varient en fonction de l'objectif du service. Seuls les événements nécessaires à cet objectif doivent donc être explicités et une logique particulière leur être associée.

Le traitement d'une réponse protocolaire varie également. Les approches généralistes associent une méthode spécifique pour le traitement des réponses. Cette approche nécessite généralement que le développeur fasse explicitement l'association avec la requête qui a généré la réponse. Au contraire, CPL capture ces événements comme un sous-traitement qui s'exécute dans la continuité de l'opération ayant relayée la requête rendant ainsi linéaire la logique de routage. Cette approche présente l'avantage de préserver le contexte du traitement de l'appel et évite la gestion explicite de l'état d'une transaction par le développeur.

Les réponses présentent également des variations. Les réponses sont caractérisées par un code de retour variant de 100 à 699. Le chiffre des centaines indique une classe de réponses. Il en existe donc six : provisoire, succès, redirection, erreur client, erreur serveur, erreur globale. À l'exception des réponses provisoires, toutes les autres terminent une transaction.

Sessions Nous avons précédemment vu que les événements sont regroupés au sein de sessions. Une session définit le cycle de vie relatif à un service, un enregistrement, un dialogue. Nous pouvons constater qu'il existe un ordre entre les sessions. Un service est ainsi présumé actif pendant que les usagers s'enregistrent. Enfin, un appel ne peut aboutir que si son destinataire est enregistré. La structure d'un service doit refléter cette *hiérarchie* entre les sessions. Cependant, la hiérarchie de sessions nécessaire pour un service donné varie en fonction de l'objectif du service. Par exemple, une session relative à l'enregistrement n'est pas toujours nécessaire et peut être omise. La session dialogue peut alors être directement déclarée dans la session service.

En-têtes Certains en-têtes du protocole SIP sont obligatoires en toutes circonstances, tandis que d'autres sont obligatoires uniquement dans certains contextes. La plupart des en-têtes sont cependant facultatifs. Un développeur doit en conséquence vérifier la présence de la plupart des en-têtes avant de les lire ou de les modifier. Le langage doit permettre aux développeurs de minimiser les opérations de contrôle. Lorsqu'un en-tête est obligatoire, son accès doit être simple et direct. En revanche, il est nécessaire d'imposer une phase de vérification de la présence des en-têtes optionnelles.

État L'état global d'un service a plusieurs niveaux de granularité. L'état global d'un service est ainsi divisé en plusieurs parties. Chaque partie respecte un cycle de vie identique aux sessions `service`, `registration`, `dialog` et `event`. Au sein de ces sessions, il est possible de définir un état lié à une transaction SIP (c.-à-d. une requête et sa réponse retournée). Cet état est alors créé pendant le traitement d'une requête puis libéré une fois la réponse associée retournée. Le langage doit faciliter et vérifier la gestion de l'état global.

7.1.3.2 Opérations

Nous venons de voir les variations qui existent sur les objets manipulés par un service de routage. Nous allons à présent voir quelles sont les variations des opérations que doit fournir le langage.

Opérations de signalisation Les événements raffinés ne sont pas tous issus de requêtes SIP. Les événements de la plate-forme qui correspondent à la session d'un service (par exemple, `deploy` ou `undeploy`) ou à la fin d'une session (par exemple, `unregister` ou `uninvite`), ne sont pas le raffinement d'une requête. Les opérations de signalisation n'ont alors pas de sens dans ce contexte particulier et doivent être proscrites.

Dans le cadre des événements raffinés issus d'une requête, plusieurs situations peuvent se présenter. Tout d'abord, si l'événement crée une session (par exemple, `REGISTER` ou `INVITE`), le service peut alors librement choisir entre le comportement par défaut qui consiste à relayer la requête vers sa destination d'origine ou une redirection vers un ou plusieurs nouveaux interlocuteurs, ou encore une redirection vers plusieurs interlocuteurs simultanément ; on parle alors de redirection en parallèle. Finalement, le service peut rejeter la requête et produire une réponse d'erreur.

Les événements intermédiaires et terminaux pour les sessions, comme les événements `ACK` ou `BYE`, ne peuvent pas tous être rejetés, en produisant une réponse d'erreur, ou routés vers une nouvelle destination. Dans les deux cas, cela mettrait les clients SIP des interlocuteurs dans un état incohérent l'un vis à vis de l'autre. Ces événements doivent donc être analysés afin de garantir qu'ils relayent bien la requête sous-jacente à son destinataire initial. Enfin, les événements indépendants des sessions (par exemple, `MESSAGE` ou `OPTIONS`) peuvent être routés comme les événements de création de sessions. Ils peuvent ainsi être relayés soit à la destination d'origine, soit vers une nouvelle destination ou des destinations multiples et simultanées, ou tout simplement rejetés par le service.

Opérations standards Les méthodes externes aux services sont soit locales au serveur d'applications, soit distantes, via des appels de méthode à distance. Parmi les opérations locales, nous pouvons citer les opérations liées à l'heure du système ou à l'accès à des fichiers locaux. Quant aux méthodes distantes, elles permettent d'intégrer, dans le processus de routage, de

l'information issue par exemple de bases de données, d'annuaires ou d'agendas. En effet, ces services sont généralement déployés sur des serveurs dédiés pour des raisons diverses (par exemple, sécurité, passage à l'échelle ou fiabilité).

7.1.4 Cahier des charges

Nous venons de voir les points communs et les variations que présente le domaine des services de routage. Il convient à présent de définir les objectifs du langage avant de définir celui-ci.

7.1.4.1 Programmation plus facile

Nous avons vu que le développement de services de routage était rarement simple avec les approches existantes. Par exemple, seul le langage CPL propose un modèle de programmation où la logique de routage est linéaire. En effet, le traitement d'un message `INVITE` déclenche l'exécution du service. Lorsqu'un nœud `proxy` est rencontré, l'exécution est suspendue jusqu'à la réception d'une réponse terminale. La programmation d'une transaction respecte ainsi un modèle de programmation impératif souvent plus simple à appréhender par un programmeur qu'un modèle événementiel comme en CCXML, en JAIN SIP ou en SIP servlets.

Nous avons également vu que la gestion de l'état d'un service était rarement simple pour un développeur. L'un des objectifs du langage doit être de simplifier la gestion de l'état. Le langage que nous concevons doit donc proposer un modèle de programmation familier, gérant automatiquement l'état lorsque cela est possible. Afin de faciliter encore un peu plus le développement, les objets et opérations du domaine que nous venons de définir, doivent être explicites et directement exploitables par les développeurs.

7.1.4.2 Réutilisation systématique

L'un des objectifs des langages dédiés est de favoriser la réutilisation du code. Notre langage permet de définir des services de haut niveau d'abstraction par rapport au système d'exploitation et au protocole SIP. Il est donc intéressant qu'il s'appuie sur un intergiciel servant d'environnement d'exécution pour les services. Cet environnement d'exécution doit permettre au langage de capitaliser sur les blocs logiciels qu'il fournit. Chaque service de routage capitalise ainsi sur cet environnement grâce à un compilateur ou un interprète pour le langage. Toute modification, ajoutant des nouvelles fonctionnalités ou des optimisations, profite alors à l'ensemble des services y compris les services déjà développés.

7.1.4.3 Fiabilité améliorée

Nous avons à plusieurs reprises mentionné l'aspect critique des communications et l'importance de préserver la plate-forme de communications, y compris en présence de services. Bien que le choix d'une architecture réseau permette en partie d'améliorer la fiabilité en séparant physiquement, par exemple, les serveurs mandataires SIP et les serveurs d'applications, il est également nécessaire de rendre fiable les services eux-mêmes. Il est ainsi possible de déployer plusieurs services sur un même serveur d'applications, tout en garantissant le bon fonctionnement de ce dernier.

Afin de garantir leur fiabilité, les services doivent faire l'objet d'analyses pour prouver qu'ils ne transgressent pas les règles imposées par le protocole. Nous avons entre autres identifié que

chaque événement lié à une requête doit conduire à une action de signalisation. La requête est soit relayée à son destinataire d'origine ou à un ou plusieurs nouveaux destinataires, soit rejetée par le service. Selon l'événement, toutes ces alternatives ne sont cependant pas possibles comme nous l'avons vu précédemment (Section 7.1.3.2).

Enfin, nous avons également vu que l'accès aux en-têtes est problématique. L'un des objectifs du langage est d'imposer aux développeurs les tests de présence des en-têtes facultatifs.

7.1.4.4 Performances

Le dernier point de notre cahier des charges consiste à fournir des performances en temps et en mémoire comparable à une implémentation manuelle. L'objectif est de réaliser, de manière automatique, les optimisations actuellement réalisées manuellement. Les opportunités d'optimisation doivent donc être détectables grâce à l'analyse des services. Un générateur de code pour le langage a ensuite la responsabilité d'appliquer les optimisations possibles. Un défi majeur consiste à minimiser en permanence l'espace mémoire nécessaire à un service. Une telle optimisation favorise le passage à l'échelle des serveurs d'applications. Dans cette optique, il faut par exemple déterminer si l'état d'un service est persistant, en totalité ou en partie, durant une transaction, une session ou l'ensemble d'un service. Il est alors possible de libérer automatiquement les composantes de l'état devenues inutiles.

7.2 Définition du langage

Une fois l'analyse de domaine réalisée, les points communs et les variations identifiés servent à définir la syntaxe du langage. Nous définissons ensuite la sémantique statique. L'analyse des services sera par la suite fondée sur cette sémantique. Nous terminons la définition du langage par la sémantique dynamique. L'objectif est alors de définir le comportement d'un service à l'exécution.

7.2.1 Syntaxe

La syntaxe détermine la structure des services et les agencements possibles des constructions du langage les unes par rapport aux autres. Un développeur définit ainsi une logique donnée par un agencement particulier.

La syntaxe du langage SPL est décrite de manière informelle, illustrée par des exemples. La définition complète en notation BNF est donnée en annexe A. Nous illustrons dans les paragraphes suivants l'utilisation des objets précédemment décrits (c.-à-d. les événements, les sessions, les en-têtes, les URI, l'état) ainsi que l'utilisation des opérations de signalisation et des opérations standards.

7.2.1.1 Événements

La figure 7.1 montre l'exemple de la redirection vers une secrétaire en cas d'échec d'un appel. Le service a été complété par un compteur du nombre de redirections ayant eu lieu pendant l'enregistrement de l'utilisateur du service, typiquement une journée. Ce compteur illustrera des fonctionnalités propres à SPL par la suite.

À chaque événement de réception d'une requête, le langage SPL fait correspondre une méthode (lignes 17 à 24). La différence notable vis à vis de solutions telles que les SIP servlets,

réside dans le fait que la méthode couvre l'intégralité d'une transaction, c'est-à-dire de la réception d'une requête par le service jusqu'à l'émission d'une réponse en retour. L'opération de signalisation `forward` (lignes 9, 18 et 21) relaie la requête associée à l'événement en cours de traitement. Une redirection de la requête est alors possible comme le montre la ligne 21. Lorsqu'une réponse est retournée, l'exécution reprend son cours. La fin du traitement d'un événement se termine par l'opération `return` qui renvoie une réponse vers l'émetteur initial (lignes 9, 21 et 23). La réponse retournée peut être produite par le service, par exemple dans le cas d'un service de filtrage d'appels. Le développeur passe alors en argument de l'opération `return` un code d'erreur SIP, par exemple `/ERROR/CLIENT/BUSY_HERE`. Il est à noter que SPL offre une vue hiérarchique des codes de retour. À l'instar de CPL, les messages provisoires ne sont pas modélisés en SPL car ils n'ont pas ou peu d'impact sur le déroulement de la communication et n'influent pas sur son aboutissement. Au premier niveau de la hiérarchie des réponses, SPL définit deux catégories, `/SUCCESS` et `/ERROR` qui correspondent respectivement aux classes de codes de retour 200 et 300 à 600. La seconde catégorie est raffinée par quatre sous-catégories : `/REDIRECTION`, `/CLIENT`, `/SERVER` et `/GLOBAL` pour chaque classe de 300 à 600. Finalement, les codes d'erreur sont organisés selon leur classe, aux feuilles de cette hiérarchie. Le code `486` est ainsi associé à la feuille `/ERROR/CLIENT/BUSY_HERE` de cette hiérarchie. L'ensemble complet de la hiérarchie est donné en annexe à la page 149 et reprend les différents codes d'erreur définis dans les RFC relatives au protocole SIP.

```
1  service sec_cpt_calls {
2    extern void log (int);
3
4    registration {
5      int cnt;
6
7      response outgoing REGISTER() {
8        cnt = 0;
9        return forward;
10     }
11
12     void unregister() {
13       log (cnt);
14     }
15
16     dialog {
17       response incoming INVITE() {
18         response r = forward;
19         if (r != /SUCCESS) {
20           cnt++;
21           return forward 'sip:secretary@example.com';
22         } else
23           return r;
24       }
25     }
26   }
27 }
```

Fig. 7.1: Service compteur en SPL

7.2.1.2 Sessions

La structure hiérarchique des sessions se retrouve dans l'inclusion des blocs de code en SPL comme l'illustre notre exemple. Nous pouvons ainsi constater que le bloc `dialog`, regroupant les événements d'un appel, est inclus dans le bloc `registration`. Ce dernier est lui-même inclus dans le bloc de méthodes `service`.

Cette structure hiérarchique prend tout son sens lorsque l'on considère l'ajout du compteur `cnt` déclaré à la ligne 5. Cette déclaration définit un état associé à la session `registration` et à ces sous-sessions, c.-à-d. la session `dialog`. La déclaration est faite à l'extérieur d'une méthode événementielle. Sa portée est donc globale au sein de la session et est disponible durant l'intégralité de l'enregistrement de l'utilisateur du service sur le réseau SIP. Le compteur est ainsi accessible depuis la méthode `unregister` (ligne 13) mais également depuis la méthode `INVITE` (ligne 20) dans la session `dialog`. L'avantage d'une telle organisation du code tient à la gestion automatique de l'état qu'elle permet. L'allocation mémoire des variables d'une session est ainsi réalisée lors du traitement de l'événement créant la session (par exemple, `REGISTER`). La sauvegarde et la restauration de l'état entre les événements sont également assurées par l'environnement d'exécution du service. Enfin, la mémoire peut être relâchée de façon sûre après l'événement de fin de session (par exemple, la libération de l'allocation du compteur après l'exécution de la méthode `unregister`).

7.2.1.3 En-têtes

Neuf en-têtes sont obligatoires dans au moins un type de requête. Nous définissons un mots-clés pour chacun d'eux. Le tableau 7.2 récapitule ces neuf mots-clés ainsi que les requêtes où ils peuvent être employés. À l'exception des mots-clés `CONTACT` et `MAX_FORWARDS`, l'accès n'est possible qu'en lecture.

Mot-clés pour les en-têtes	Requêtes où l'en-tête est obligatoire
`CALL_ID`	*toutes*
`CONTACT`	`INVITE`, `SUBSCRIBE`, `NOTIFY`
`CSEQ`	*toutes*
`EVENT`	`SUBSCRIBE`, `NOTIFY`, `PUBLISH`
`FROM`	*toutes*
`MAX_FORWARDS`	*toutes*
`SUBSCRIPTION_STATE`	`NOTIFY`
`TO`	*toutes*
`VIA`	*toutes*

Tab. 7.2: En-têtes accessibles par mots-clés

L'exemple de la figure 7.2 permet à son utilisateur de ne prendre que les appels provenant de sa femme en vérifiant la valeur de l'en-tête `FROM` (lignes 5 et 6). Sinon l'appel est rejeté avec le code `BUSY_HERE` (ligne 13). Dans ce cas, la présence d'un en-tête précisant le sujet de l'appel est vérifiée (construction `when`, ligne 10). Si le sujet de l'appel est présent, sa valeur est accessible via la variable `sub` déclarée à la ligne 10. Un nouveau sujet est construit en concaténant à la chaîne de caractères `"[Missed Call]"` l'éventuel sujet actuel (lignes 9

et 11). La réponse retournée est personnalisée avec une raison particulière (ligne 14), l'adresse de courrier électronique de l'utilisateur (ligne 15) et le sujet précédemment construit (ligne 16).

```
1  service header_field {
2    dialog {
3      response incoming INVITE() {
4
5        if (FROM == 'sip:wife@example.com') {
6          return forward;
7        }
8        else {
9          string obj = "[Missed Call]";
10         when this (subject: string sub) {
11           obj += sub;
12         }
13         return /ERROR/CLIENT/BUSY_HERE
14           with { reason = "I'm not available. Please, leave me a message.",
15             contact: 'mailto:me@example.com',
16             subject: obj
17           };
18       }
19     }
20   }
21 }
```

FIG. 7.2: Service de manipulation d'en-têtes en SPL

En plus de l'accès par mot-clé pour les en-têtes obligatoires, SPL fournit les constructions `when` (ligne 10) et `with` (ligne 14) qui permettent d'accéder aux autres en-têtes respectivement en lecture et écriture aussi bien pour les requêtes que pour les réponses.

7.2.1.4 Uniform Ressources Identifier

Afin de pouvoir raisonner sur les URI, celles-ci sont différenciées des chaînes de caractères par l'utilisation du caractère `'` en lieu et place du caractère `"` qui délimite les chaînes de caractères. L'exemple précédent (Figure 7.2) montre l'utilisation des URI et des chaînes de caractères (lignes 5, 9, 14 et 15). Les URI sont ainsi clairement identifiées et leur analyse est possible statiquement.

7.2.1.5 État

Le langage SPL offre deux mécanismes complémentaires pour faciliter la gestion de l'état. Le premier facilite la gestion de l'état au sein d'une méthode événementielle. Le second facilite la gestion de l'état entre les méthodes.

État intra-méthode L'opération de signalisation `forward` interrompt l'exécution de la méthode en cours et émet la requête courante. L'état du service est alors sauvegardé. Il est restauré lorsqu'une réponse à la requête est reçue.

La figure 7.3 illustre ce comportement pour un appel entrant et le traitement de la méthode `INVITE`. La réception de la requête déclenche l'exécution de la méthode correspondante dans le service. L'opération `forward` interrompt l'exécution, sauvegarde l'état du service et transmet la requête vers le destinataire de l'appel. Lorsque celui-ci répond, en décrochant ou en refusant

l'appel, sa réponse restaure l'état du service qui reprend son exécution. L'opération `return` renvoie finalement une réponse à l'émetteur de l'appel.

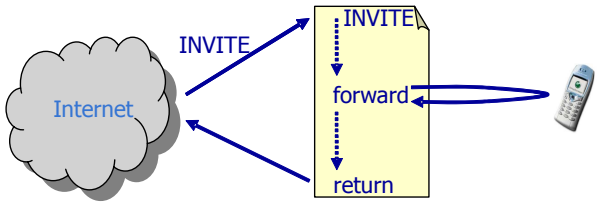


FIG. 7.3: Flot de contrôle intra-méthodes

État inter-méthode Une session représente un cycle de vie. Ce cycle impose un certain ordre entre les événements. Un flot de contrôle entre les événements est ainsi défini par le protocole. Dans le cas d'un appel, nous avons par exemple la séquence `INVITE`, `ACK`, `BYE`. Il est intéressant pour un développeur de raffiner ce flot de contrôle en fonction d'éléments connus dynamiquement. Il est ainsi possible de préciser un comportement particulier pour les méthodes qui suivent la requête `INVITE`. Afin de simplifier le raffinement, SPL fournit la construction `branch`. Elle permet de passer des informations sur le flot de contrôle d'une méthode à la suivante.

Les branches sont sélectionnées par le développeur à la fin d'une méthode. L'environnement conserve alors l'information sur la branche sélectionnée et l'associe à la session. Une branche particulière, nommée `default`, est initialement sélectionnée. Quand une méthode d'une session service ou enregistrement est invoquée, le code de la branche courante est exécuté ou celui de la branche par défaut si aucune branche déclarée ne correspond. La branche par défaut permet alors un traitement commun à une ou plusieurs branches. Les sessions d'appels et de souscription sont traitées de manière similaire, excepté que la nouvelle branche hérite de la branche actuelle au lieu de la remplacer. Une pile permet de mémoriser les branches héritées et d'ajouter la nouvelle branche. Lors de l'invocation d'une méthode, le code correspondant à la plus récente branche dans la pile est exécuté.

Ce mécanisme est illustré à la figure 7.4 par l'extrait d'un service. L'utilisateur est responsable d'un secrétariat accessible via l'URI `sip:secretary@example.com` qui est un alias pour sa propre URI. Les appels entrants sont catégorisés comme étant destinés au secrétariat (branche `secretary`, ligne 10), provenant d'un proche de l'utilisateur (branche `private`, ligne 14). Les autres appels ne sont pas différenciés et conservent la branche par défaut (ligne 18). Cette information contextuelle est automatiquement associée à la session à la fin de la méthode `INVITE`. Elle est restaurée lors de la réception de l'événement `ACK` (lignes 22 et 25). La branche par défaut est également exécutée si la branche est marquée comme privée car aucun traitement particulier n'est déclaré pour le type de branche `private`. Enfin, lorsque l'un des interlocuteurs raccroche, l'une des trois branches est exécutée en fonction de la branche active (lignes 28 à 30).

```
service branch_example {
  [...]
  dialog {
   [...]
    response incoming INVITE() {
      response resp = forward;
      if (TO == 'sip:secretary@example.com') {
        [...]
        // Session tagged as "secretary"
        return resp branch secretary;
      } else if(FROM == [...]){
        [...]
        // Session tagged as "private"
        return resp branch private;
      } else {
        [...]
        // Session keep its default tag
        return resp;
      }
    }
    response incoming ACK() {
      branch secretary { // Specific treatment for secretary sessions
        [...]
      }
      branch default {[...]}  // Specific treatment for private and default sessions
    }
    response BYE() {
      branch secretary { [...] }
      branch private { [...] }
      branch default { [...] }
    }
  }
}
```

FIG. 7.4: Flot de contrôle inter-méthodes

7.2.1.6 Opérations de signalisation

SPL fournit deux opérations de signalisation, une pour les requêtes et une pour les réponses. L'opération `forward` sert à relayer les requêtes. Une URI peut être fournie à cet opérateur afin de modifier le routage de la requête courante. Dans le cas d'une redirection vers plusieurs destinataires, une séquence d'URI est fournie au lieu d'une unique URI. L'ensemble des destinataires peut être contacté soit l'un après l'autre, jusqu'à obtenir une réponse positive, soit simultanément. Le modificateur `parallel` préfixe alors l'expression `forward`. Quelque soit l'utilisation de l'opérateur, il retourne une unique réponse finale, c'est-à-dire une réponse dont le code de retour est supérieur ou égal à 200. Dans le cas d'une redirection vers plusieurs utilisateurs, plusieurs réponses sont retournées, l'une des meilleures est alors retenue comme valeur de l'expression `forward`. Une réponse est meilleure qu'une autre si elle est plus précise ou qu'elle permet un progrès dans la communication. Une réponse 501, indiquant qu'une fonction n'est pas implémentée sur un serveur, est par exemple plus précise que l'erreur 500, indiquant une erreur interne au serveur. De même, une réponse suggérant une redirection, est meilleure qu'une réponse avec un signal d'occupation car le serveur peut utiliser la redirection pour déterminer un nouveau destinataire. Enfin, un signal d'occupation est lui-même meilleur qu'une réponse signalant un utilisateur inconnu ou une erreur interne sur un serveur, car il suggère de rappeler ultérieurement. Si les réponses sont équivalentes, l'une d'entre elles est

choisie de manière non déterministe.

La seconde opération, `return`, consiste à retourner une réponse vers l'émetteur de la requête initiale. Cette réponse peut soit provenir d'un précédent `forward`, soit d'une réponse finale constante déclarée dans le service. Afin de faciliter la production de réponse d'erreur, l'ensemble des réponses possibles est fourni par le langage. Un développeur peut ainsi naviguer dans la hiérarchie des codes de retour de manière symbolique, par exemple `/ERROR/CLIENT/BUSY_HERE`. Il est également possible d'utiliser cette hiérarchie de façon partielle, par exemple `/ERROR/CLIENT/` ou même `/ERROR`. Un développeur peut ainsi très facilement retourner une erreur ou comparer une réponse reçue avec une classe d'erreur.

7.2.1.7 Opérations standards

En plus des opérations arithmétiques et logiques de bases, le langage SPL permet d'invoquer des méthodes extérieures aux services. Ces méthodes doivent avoir été préalablement chargées et être disponibles dans l'environnement d'exécution. Dans l'exemple SPL de redirection vers la secrétaire (Figure 7.1, page 71), un module de gestion de journal est chargé (ligne 2). Il est invoqué par la suite (ligne 13) afin d'enregistrer la valeur du compteur. Nous pouvons noter le modificateur `extern` lors de la déclaration. Ce modificateur indique que la méthode est extérieure au service

7.2.2 Sémantique statique

Afin de déceler les services syntaxiquement corrects mais sémantiquement faux, nous définissons plusieurs analyses. Ces analyses se concentrent sur la signalisation. Nous définissons en conséquence une syntaxe abstraite du langage SPL qui se concentre sur les aspects de signalisation. Les opérations de signalisation ne pouvant avoir lieu que dans les méthodes d'événements issus de requêtes SIP, la définition de la syntaxe se limite à ces méthodes et omet les sessions. Seules les déclarations de variables de type `response` sont également considérées. Nous appelons cette syntaxe abstraite SPL_{fwd} (Figure 7.5).

$$
\begin{array}{lcl}
H & ::= & \texttt{response}\ DIR_{opt}\ method\_name\ \{\ D^*\ S\ \} \\
DIR_{opt} & := & \texttt{None} \mid \texttt{Some}(DIR) \\
DIR & ::= & \texttt{incoming} \mid \texttt{outgoing} \\
D & ::= & \texttt{response}\ x\texttt{;} \\
S & ::= & id\ \texttt{=}\ \texttt{fwd}\ URI_{opt} \mid \texttt{return}\ E_R \mid \epsilon \\
 & \mid & \texttt{cond}(E_B\texttt{,}\ S_1\texttt{,}\ S_2) \mid S_1\texttt{;}\ S_2 \\
URI_{opt} & := & \texttt{None} \mid \texttt{Some}(E_{URI}) \\
E_R & ::= & id \mid \texttt{/ERROR} \mid \texttt{/SUCCESS} \\
E_B & ::= & id\ \texttt{==}\ E_R \\
E_{URI} & ::= & URI \mid \langle URI, \ldots \rangle \\
URI & ::= & \texttt{'}constant\texttt{'}
\end{array}
$$

FIG. 7.5: Syntaxe abstraite du langage SPL_{fwd}

Un programme SPL peut directement être réécrit en son homologue SPL_{fwd}. La figure 7.6

illustre cette réécriture sur la méthode `INVITE` du service de redirection vers la secrétaire en cas de non réponse. De plus, nous supposerons que des techniques de transformations de programmes telles que la propagation de copies et la propagation de constantes [ASU86] sont appliquées afin d'obtenir le programme $SPL_{fwd\cdot}$

```
1 response Some(incoming) INVITE {
2   response r;
3   r = fwd None;
4   cond (r == /ERROR,
5         r = fwd Some('sip:secretary@example.com'); return r,
6         return r)
7 }
```

FIG. 7.6: Méthode `INVITE` du service compteur en $SPL_{fwd\cdot}$

Nous présentons tout d'abord l'analyse vérifiant qu'un service est bien typé. Nous verrons ensuite l'analyse de l'opération `forward` permettant de détecter les services susceptibles de perdre une réponse positive émise lorsque l'interlocuteur décroche.

7.2.2.1 Analyse de type

Nous nous concentrons dans cette section sur les opérations de signalisation. L'intégralité de la sémantique statique vérifiant les types d'un programme SPL est disponible en annexe B, page 151. La figure 7.7 donne les règles de type pour le langage $SPL_{fwd\cdot}$

Les opérations de signalisation ne peuvent être utilisées que dans les méthodes événementielles. La règle 7.1 définit la vérification de type pour une telle méthode. L'évaluation des déclarations puis de l'instruction doit produire un type conforme à celui de la méthode, c'est-à-dire de type `response`[1]. Les règles 7.2 et 7.3 définissent les règles de type pour les déclarations. La déclaration d'une nouvelle variable de type `response` associe son identifieur au type `response` dans l'environnement τ. Une fois l'ensemble des déclarations évaluées, les instructions de la méthode sont évaluées vis à vis de l'environnement τ.

La vérification des instructions est décrite par les règles 7.4 à 7.9. Seule l'instruction `return` produit un type de retour `response`. La règle 7.9 impose qu'il s'agisse de la dernière instruction. Enfin, la règle 7.8 impose que les deux branches des conditionnelles retournent une valeur de même type. Ainsi, si une branche retourne une valeur de type réponse, cette règle garantit qu'une réponse est également retournée par l'autre branche. Les règles 7.4 et 7.5 décrivent l'opération de signalisation `forward` respectivement sans argument et avec argument. Si un argument est présent, il doit être de type URI ou liste d'URI (Règle 7.5). Dans les deux cas, l'instruction produit un type `void` et la variable qui reçoit la valeur du `forward` doit être définie dans l'environnement et être de type `response`.

Les règles 7.10 à 7.13 permettent de vérifier les expressions. Pour une variable, son type est consulté dans l'environnement τ (Règle 7.10). Les constantes produisent leur type respectif (Règles 7.11, 7.13 et 7.14). Enfin, le résultat de la comparaison de deux valeurs est un booléen. De plus, les deux valeurs comparées doivent être de type identique.

Nous pouvons garantir que tout programme respectant l'ensemble des règles de la figure 7.7 produit une unique réponse pour chacune des méthodes événementielles qu'il comporte. De

[1] à l'exception de la requête `ACK` qui appartenant à la transaction initial a un type de retour `void`

plus, seules les expressions de type uri ou uri list peuvent être utilisées comme opérande de l'opération `forward`.

$$\frac{\varnothing \vdash^{D} D : \tau_D \qquad \tau_D \vdash^{S} S : \mathsf{response}}{\vdash^{H} \texttt{response}\ \ dir^{?}\ \ method\_name\ \ \{\ \ D\ \ S\ \ \} : \mathsf{response}} \tag{7.1}$$

$$\frac{\tau' := \tau[id \mapsto \mathsf{response}]}{\tau \vdash^{D} \texttt{response}\ \ id : \tau'} \tag{7.2}$$

$$\frac{\tau \vdash^{D} D_1 : \tau_1 \qquad \tau_1 \vdash^{D} D_2 : \tau_2}{\tau \vdash^{D} D_1\ \texttt{;}\ \ D_2 : \tau_2} \tag{7.3}$$

$$\frac{\tau(id) = \mathsf{response}}{\tau \vdash^{S} id\ \texttt{=}\ \texttt{fwd} : \mathsf{void}} \tag{7.4}$$

$$\frac{\tau \vdash^{E_{URI}} URI : \mathsf{uri} \vee \mathsf{uri\ list} \qquad \tau(id) = \mathsf{response}}{\tau \vdash^{S} id\ \texttt{=}\ \texttt{fwd}\ \ URI : \mathsf{void}} \tag{7.5}$$

$$\frac{\tau \vdash^{E_R} E_R : \mathsf{response}}{\tau \vdash^{S} \texttt{return}\ \ E_R : \mathsf{response}} \tag{7.6}$$

$$\frac{}{\tau \vdash^{S} \epsilon : \mathsf{void}} \tag{7.7}$$

$$\frac{\begin{array}{c}\tau \vdash^{E_B} E_B : \mathsf{bool} \\ \tau \vdash^{S} S_1 : t_1 \\ \tau \vdash^{S} S_2 : t_2 \\ t_1 = t_2\end{array}}{\tau \vdash^{S} \texttt{cond}\ \ \texttt{(}E_B\texttt{,}\ \ S_1\texttt{,}\ \ S_2\texttt{)} : t_1} \tag{7.8}$$

$$\frac{\tau \vdash^{S} S_1 : \mathsf{void} \qquad \tau \vdash^{S} S_2 : t}{\tau \vdash^{S} S_1\ \texttt{;}\ \ S_2 : t} \tag{7.9}$$

$$\frac{}{\tau \vdash^{E_R} id : \tau(id)} \tag{7.10}$$

$$\frac{c \in \{\texttt{/SUCCESS}, \texttt{/ERROR}\}}{\tau \vdash^{E_R} c : \mathsf{response}} \tag{7.11}$$

$$\frac{\begin{array}{c}\tau \vdash^{E_R} E_R : t_1 \\ \tau \vdash^{E_R} id : t_2 \\ t_1 = t_2\end{array}}{\tau \vdash^{E_B} id\ \texttt{==}\ E_R : \mathsf{bool}} \tag{7.12}$$

$$\frac{c \in \{\texttt{'sip:doe@example.com'}, \ldots\}}{\tau \vdash^{E_{URI}} c : \mathsf{uri}} \tag{7.13}$$

$$\frac{c \in \{\langle \texttt{'sip:doe@example.com'}, \ldots\rangle, \ldots\}}{\tau \vdash^{E_{URI}} c : \mathsf{uri\ list}} \tag{7.14}$$

FIG. 7.7: Vérification des types pour le langage SPL_{fwd}

7.2.2.2 Analyse de l'opération `forward`

Nous venons de voir l'analyse de type de programmes SPL. Cependant, cette analyse, bien que nécessaire, n'est pas suffisante. Elle ne garantit pas qu'une éventuelle réponse positive, reçue grâce à un `forward`, est bien retransmise à l'appelant. Nous présentons dans cette section une analyse supplémentaire permettant d'interdire une nouvelle opération `forward` lorsqu'une précédente a produit une réponse positive, c.-à-d. `/SUCCESS`. Cette restriction

permet d'empêcher un service de réaliser une redirection d'appels vers quelqu'un si une autre personne a déjà accepté l'appel.

La règle 7.23 évalue une méthode et a la forme suivante :

$$\vdash^{H} \texttt{response}\ \ dir^{?}\ \ method\_name\ \ \{\ \ D\ \ S\ \ \}\ :\ \mathfrak{P}$$

Cette règle retourne une valeur $\mathfrak{P}$ qui indique si la méthode évaluée est valide vis à vis de la propriété que nous considérons. Cette valeur devient fausse aussitôt qu'une utilisation illégale de l'opération `forward` est détectée.

Une règle de la forme $\tau \vdash^{E_R} exp : \sigma$ (Règles 7.24 à 7.26) signifie que l'évaluation de l'expression *exp* dans l'environnement τ retourne le statut σ. Le statut σ a pour valeur success si l'expression est connue pour être une réponse avec un code de retour de type `2xx`. σ a pour valeur error si l'expression est connue pour être une réponse avec un code de retour supérieur ou égal à 300. Enfin, σ a pour valeur $\bot$ si le code de retour de la réponse n'est pas connu. Ces valeurs de statut forment un ordre partiel tel que $\bot \sqsubseteq$ success et $\bot \sqsubseteq$ error. Cet ordre partiel sert lors de la fusion des environnements réalisée à la règle 7.21.

La comparaison entre deux réponses (Règle 7.27) retourne une paire constituée d'un identifieur et d'un statut. Cette comparaison permet de déterminer si la réponse décrite par l'identifieur est un succès ou une erreur. La découverte de cette information est ensuite propagée dans chacune des branches de la conditionnelle (Règle 7.21).

Le point clé de cette analyse est dans le traitement du `forward` (Règles 7.17 et 7.18) et de la conditionnelle. En effet, la réponse reçue par une opération `forward` doit être systématiquement sauvegardée dans une variable. L'environnement τ modélise ainsi les effets de l'ensemble des opérations `forward` qui ont eu lieu. Si cet environnement contient une variable dont la valeur est success ou $\bot$, alors un précédent `forward` a ou peut avoir réussi. Dans ces deux cas, une nouvelle opération `forward` est prohibée et la propriété de l'analyse prend la valeur fausse (Règle 7.17). Si toutes les variables de l'environnement sont connues comme ayant la valeur error (Règle 7.18), nous avons l'assurance qu'aucun précédent `forward` n'a réussi et qu'un nouveau est possible. Dans tous les cas, l'environnement est mis à jour et la variable *id* est associée à la valeur $\bot$, indiquant que la nouvelle valeur de l'opération `forward` n'a pas encore été testée et qu'il peut donc s'agir d'un succès.

Le test d'une variable est réalisé par la règle de la conditionnelle (Règle 7.21) en coopération avec les règles 7.24 à 7.27. La branche S_1, correspondant au *then*, est analysée sachant que l'environnement reflète le fait que le test était positif. La branche S_2, *else*, est analysée sachant que le test était négatif. Dans ce cas, l'environnement est construit avec l'opérateur $\neg$, définit comme suit, $\neg$success $=$ error, $\neg$error $=$ success et $\neg\bot = \bot$. L'évaluation des branches S_1 et S_2 conduit à définir deux nouveaux environnement, τ_1 et τ_2. Comme chacun d'eux peut être disponible à l'exécution, l'analyse les fusionne pour évaluer les instructions suivantes en utilisant l'opérateur $\uplus : \tau \times \tau \rightarrow \tau$, défini comme suit :

$$\tau_1 \uplus \tau_2 \quad = \quad \{(x, \sigma_1 \sqcap \sigma_2) \mid (x, \sigma_1) \in \tau_1 \wedge (x, \sigma_2) \in \tau_2\}$$

Cet opérateur garantit que si la réponse d'un `forward` n'est pas connue dans l'une des deux branches, ou possède des valeurs différentes dans chacune, alors la variable correspondante prend la valeur $\bot$.

$$\frac{\tau' := \tau[id \mapsto \mathsf{error}]}{\tau \vdash^D \texttt{response}\ \ id : \tau'} \quad (7.15) \qquad \frac{\tau \vdash^D D_1 : \tau_1 \qquad \tau_1 \vdash^D D_2 : \tau_2}{\tau \vdash^D D_1\ \texttt{;}\ \ D_2 : \tau_2} \quad (7.16)$$

$$\frac{\begin{array}{c}\exists (x,\sigma) \in \tau.\ \sigma \neq \mathsf{error} \\ \tau' := \tau[id \mapsto \bot]\end{array}}{\tau \vdash^S id\ =\ \texttt{fwd}\ \ URI^? : \langle \tau', \mathsf{false} \rangle} \quad (7.17) \qquad \frac{\begin{array}{c}\forall (x,\sigma) \in \tau.\ \sigma = \mathsf{error} \\ \tau' := \tau[id \mapsto \bot]\end{array}}{\tau \vdash^S id\ =\ \texttt{fwd}\ \ URI^? : \langle \tau', \mathsf{true} \rangle} \quad (7.18)$$

$$\frac{}{\tau \vdash^S \texttt{return}\ \ E_R : \langle \tau, \mathsf{true} \rangle} \quad (7.19) \qquad \frac{}{\tau \vdash^S \epsilon : \langle \tau, \mathsf{true} \rangle} \quad (7.20)$$

$$\frac{\begin{array}{c}\tau \vdash^{E_B} E_B : (id, \sigma) \\ \tau[id \mapsto \sigma] \vdash^S S_1 : \langle \tau_1, fwd_1 \rangle \\ \tau[id \mapsto \neg\sigma] \vdash^S S_2 : \langle \tau_2, fwd_2 \rangle\end{array}}{\tau \vdash^S \texttt{cond}\ \ (E_B\texttt{,}\ \ S_1\texttt{,}\ \ S_2) : \langle \tau_1 \uplus \tau_2, fwd_1 \wedge fwd_2 \rangle} \quad (7.21)$$

$$\frac{\begin{array}{c}\tau \vdash^S S_1 : \langle \tau_1, fwd_1 \rangle \\ \tau_1 \vdash^S S_2 : \langle \tau_2, fwd_2 \rangle\end{array}}{\tau \vdash^S S_1\ \texttt{;}\ \ S_2 : \langle \tau_2, fwd_1 \wedge fwd_2 \rangle} \quad (7.22)$$

$$\frac{\varnothing \vdash^D D : \tau_D \qquad \tau_D \vdash^S S : \langle \tau, fwd \rangle}{\vdash^H \texttt{response}\ \ dir^?\ \ method\_name\ \ \{\ \ D\ \ S\ \ \}} : fwd \quad (7.23)$$

$$\frac{(id, \sigma) \in \tau}{\tau \vdash^{E_R} id : \sigma} \quad (7.24) \qquad \frac{}{\tau \vdash^{E_R} \texttt{/SUCCESS} : \mathsf{success}} \quad (7.25)$$

$$\frac{}{\tau \vdash^{E_R} \texttt{/ERROR} : \mathsf{error}} \quad (7.26) \qquad \frac{\tau \vdash^{E_R} E_R : \sigma}{\tau \vdash^{E_B} id\ ==\ E_R : (id, \sigma)} \quad (7.27)$$

FIG. 7.8: Sémantique statique du langage SPL_{fwd}

7.2.2.3 Analyse de l'expression constante `/SUCCESS`

L'analyse des occurrences de l'expression constante `/SUCCESS` permet de garantir qu'un service de routage ne fabrique pas de réponse de succès. Cette propriété est importante pour garantir la cohérence de l'état des périphériques utilisateurs. L'occurrence de cette constante est réservée à l'expression des conditionnelles. Cette constante ne peut pas servir à définir une variable de type réponse et n'est pas une valeur de retour valide pour l'instruction `return`.

7.2.3 Sémantique dynamique

Nous venons de donner des analyses permettant de garantir qu'une réponse est bien produite pour chaque requête SIP reçue et qu'aucune nouvelle opération `forward` n'est réalisable lorsqu'une réponse ayant, ou pouvant avoir, un code de retour de type `2xx` est déjà en cours de traitement dans la méthode. Ces analyses sont réalisées statiquement, c'est-à-dire avant l'exécution, afin que seuls les services valides puissent être déployés dans l'environnement d'exécution. Nous allons à présent spécifier le comportement des services à l'aide d'une sémantique dynamique. Celle-ci permettra de documenter le langage et de définir des outils pour l'exécution de services. La présentation de la sémantique dynamique se limitera aux aspects

relatifs aux sessions hiérarchiques et à la signalisation dans les méthodes ; l'intégralité de la sémantique dynamique est disponible en annexe (Section B.3 et suivantes, page 155). Il est important de noter que cette section est indépendante de la précédente et que certains noms de variable comme σ ou τ sont réutilisés sans lien avec la section précédente.

7.2.3.1 Sessions

Nous avons vu que SPL est conçu autour de la notion de sessions organisées de manière hiérarchique. Une session comprend un ensemble de variables, c.-à-d. un état, et des méthodes (Sections 7.1.2.1, 7.1.3.1 et 7.2.1.2). Afin d'identifier de manière unique une session, chacune est associée à un unique *label*. Pour décrire la position d'une session dans la hiérarchie, une *adresse* est également associée à une session. L'adresse d'une session est formée par la séquence des labels de la session et des sessions parentes. L'état d'une session est stocké dans un état global σ. Cet état global associe à une adresse de session un tuple contenant la branche active, qui est représentée par une liste d'identifieurs, le statut de la session, l'environnement de la session qui associe les variables d'une session à leur valeur, et la liste des adresses des sous-sessions :

$$\sigma \in \mathsf{state} = \mathsf{address} \rightarrow \mathsf{branch\ list} \times \mathsf{status} \times \mathsf{env} \times \mathsf{address\ list}$$

Le statut d'une session, de type status, est composé de quatre informations : *live*, $persistent_{\perp}$, $close\_fn_{\perp}$ et *open*. *live* est un marqueur booléen indiquant que la session est active lorsque le marqueur est vrai. $persistent_{\perp}$ est un marqueur booléen indiquant la persistance de la session. Il peut être non initialisé et alors avoir la valeur $\perp$. $close\_fn_{\perp}$ est le nom de la méthode à invoquer lors de la fermeture d'une session. La valeur $\perp$ dénote l'absence d'une telle méthode. *open* est un entier qui indique le nombre de transactions en cours au niveau de la session courante.

$$\mathsf{status} = \mathsf{bool} \times \mathsf{bool}_{\perp} \times \mathsf{string}_{\perp} \times \mathsf{int}$$

La sémantique de SPL utilise un ensemble de fonctions qui manipulent les sessions : create_session, prepare_method_invocation, continue_session, et end_session.

La fonction create_session étend l'état global σ avec une entrée pour une nouvelle session. Cette fonction a la signature suivante :

$$\mathsf{create\_session} : \mathsf{program} \times \mathsf{state} \times \mathsf{address} \times \mathsf{method\_name} \rightarrow \mathsf{state}$$

Lors de la création d'une session, le statut initial de la session est défini comme suit : *live* a la valeur vrai, $persistent_{\perp}$ la valeur $\perp$ et *open* la valeur zéro. La liste des sous-sessions est initialisée à une liste vide. La valeur $close\_fn_{\perp}$ et la valeur de la branche active dépendent du type de session. Les valeurs sont indiquées au tableau 7.3. L'environnement de session est initialisé avec l'évaluation des déclarations relatives à la session. Finalement, la fonction create_session retourne un nouvel environnement global qui contient la nouvelle session.

Une fois la nouvelle session créée, les méthodes de la session peuvent être invoquées. L'invocation est initialisée par la fonction prepare_method_invocation du type :

$$\mathsf{prepare\_method\_invocation} : \mathsf{program} \times \mathsf{state} \times \mathsf{address} \times \mathsf{method\_name} \times \mathsf{direction} \rightarrow \mathsf{decl\ list} \times \mathsf{stmt} \times \mathsf{env\ list} \times \mathsf{state}$$

Type de session	Fonction finale	Branches
`service`	`undeploy`	`default`
`registration`	`unregister`	`default`
`dialog`	`uninvite`	liste héritée
`event`	`unsubscribe`	liste héritée

TAB. 7.3: Configuration d'une session en fonction de son type

Cette fonction extrait les déclarations et les instructions associées à la méthode, récupère l'ensemble des environnements dont hérite la session et incrémente le compteur *open* pour indiquer qu'une transaction est en cours. La listes des déclarations, les instructions, l'environnement de la session et un nouvel état global sont alors retournés. L'environnement de la session est formé d'une séquence d'environnements. Chacun d'eux correspond à un niveau dans la hiérarchie des sessions.

L'exécution des instructions manipule la séquence des environnements de session précédemment obtenue. Lorsque l'exécution se termine, ces environnements doivent être mis à jour dans l'environnement global σ. Cette opération est dévolue à la fonction continue_session ayant le type suivant :

$$\mathsf{continue\_session} : \mathsf{program} \times \mathsf{state} \times \mathsf{address} \times \mathsf{env\ list} \rightarrow \mathsf{state}$$

Le comportement de cette fonction dépend de l'état du marqueur *live* de la session. Si la session est active, σ est mis à jour avec les nouveaux environnements produits par l'exécution de la méthode, et le compteur *open* est diminué d'une transaction active. Si la session n'est plus active, la fonction end_session est invoquée et détermine si la session doit être détruite en fonction du statut de la session. La fonction se termine en retournant un nouvel état σ du système.

$$\mathsf{end\_session} : \mathsf{program} \times \mathsf{state} \times \mathsf{address} \rightarrow \mathsf{state}$$

Le partage de l'état, entre les méthodes et les sessions, impose une gestion automatique de cet état. La terminaison d'une session est une phase importante et délicate de cette gestion. La terminaison d'une session peut avoir lieu pour plusieurs raisons : une terminaison normale (par exemple, un interlocuteur raccroche), la création a échoué (par exemple, personne n'a répondu à l'appel) ou une alarme protocolaire invoque la méthode de clôture de session (par exemple, une erreur réseau est survenue). Néanmoins, il n'est pas toujours désirable de détruire une session immédiatement. Il est par exemple nécessaire d'attendre que les méthodes en cours ou les sous-sessions actives reçoivent une réponse et se terminent. Une session n'est donc détruite que si son compteur *open* est nul et qu'il n'y a donc aucune méthode en attente d'une réponse.

La hiérarchie des sessions impose également une gestion particulière lors de la clôture des sessions. Du point de vue du protocole SIP, une session peut être terminée mais, du point de vue du langage SPL, son état doit parfois persister. C'est notamment le cas d'une session enregistrement qui peut expirer à la demande de l'utilisateur ou lorsque l'alarme correspondante expire. Dans ce cas, il n'est pourtant pas nécessaire de clore les sous-sessions `dialog` et `event` éventuellement actives. La session `registration` doit alors être persistante afin que les variables associées puissent être référencées par les sous-sessions. Le marqueur $persistent_{\perp}$ indique alors si une session doit attendre la fin des sessions filles ou si ces dernières doivent

être closes immédiatement, en même temps que la session courante. La méthode `unregister` est ainsi persistante afin que les dialogues en cours puissent se terminer normalement alors que la méthode `undeploy` est non-persistante afin que l'administrateur d'un système puisse éteindre un serveur sans délai.

La gestion automatique des sessions par le langage permet libérer de manière systématique les allocations mémoires liées à une session particulière lorsque la session est close. De plus, la structure hiérarchique des sessions et la notion de persistance permettent au système de libérer également l'état des éventuelles sous-sessions. L'état alloué est ainsi minimal à tout instant de l'exécution des services.

7.2.3.2 Opérations de signalisation

Nous venons de voir la sémantique dynamique pour les sessions hiérarchiques dans SPL. Nous allons à présent voir un second concept clé du langage, la sémantique dynamique des opérations de signalisation. Nous verrons tout d'abord l'invocation de méthodes lors de la réception d'une requête puis l'opération `forward` qui relaie les requêtes et retourne la réponse finale correspondante.

Invocation de méthodes La sémantique des termes de SPL est décrite à l'aide d'une machine abstraite fondée sur les continuations [ADM05]. L'exécution de cette machine est spécifiée comme une séquence de configurations. La première de ces configurations correspond à la réception d'une requête SIP. La dernière correspond à l'émission d'une réponse du service vers la machine virtuelle SIP. Les configurations intermédiaires représentent l'exécution d'un terme du langage ou d'une continuation. Une continuation est similaire à la pile de contrôle utilisée dans l'implémentation des langages impératifs. A chaque évaluation d'un terme par la sémantique, une tâche est ajoutée à la continuation courante. Cette tâche contient tous les éléments nécessaires pour terminer l'exécution du terme courant. Cette approche rend chaque continuation autonome et est exploitée lors de la sémantique du terme `forward`. Les configurations décrivent une sémantique à petit pas de l'invocation d'une méthode. Il existe quatre types de configurations : l'invocation de méthodes, l'évaluation du corps d'une méthode, la continuation d'une méthode, c.-à-d. après une opération `forward`, et le retour d'une méthode.

- Invocation d'une méthode : $message, \phi, \sigma \models_{\text{mi}} service$
- Évaluation du corps d'une méthode : $\langle \sigma, address \rangle, envs, s \models_{\text{h}} decls, stmt$
- Continuation d'une méthode : $\langle \sigma, address \rangle, \langle envs, uri, local\_env \rangle \models_{\text{mc}} s, resp$
- Retour d'une méthode vers la machine virtuelle SIP : $value, \sigma$

L'évaluation d'un terme par une règle sémantique réécrit la configuration courante et en produit une nouvelle. Cette réécriture consiste essentiellement à empiler et dépiler des tâches sur la pile de contrôle, c'est-à-dire modifier la continuation. Nous illustrons ici l'invocation de la méthode `INVITE` qui crée une nouvelle session `dialog`. Le traitement des autres méthodes est similaire.

La règle 7.28 initialise l'invocation de la méthode `INVITE`. Après avoir récupéré la branche active dans la session parente, une nouvelle session est créée dans l'environnement global σ ; puis, l'exécution des déclarations et des instructions est préparée par la méthode prepare_method_invocation. La règle produit alors une nouvelle configuration permettant l'exécution des déclarations et des instructions pour la branche active. La continuation est alors

armée avec la tâche `INITIAL_INVITE`.

$$\frac{\begin{array}{c} address = \langle service, rid, did \rangle \\ \mathsf{lookup\_branches}(\sigma, \mathsf{parent}(address)) = \langle branch \rangle \\ \mathsf{create\_session}(\phi, \sigma, address, \langle branch \rangle, \texttt{undialog}) = \sigma' \\ \mathsf{prepare\_method\_invocation}(\phi, \sigma', direction, \texttt{INVITE}) = \mu \\ \mu = \langle m, decls, stmt, envs, \sigma'' \rangle \\ \tau = \langle \sigma'', address \rangle \qquad r = \langle envs, \langle m, \langle rq, headers \rangle \rangle \rangle \end{array}}{\begin{array}{r} \langle \texttt{INVITE}(rid, did), direction, rq, headers \rangle, \phi, \sigma \models_{\text{mi}} service \\ \Rightarrow \tau, r, \langle \texttt{INITIAL\_INVITE}\ \ \phi \rangle \models_{\text{h}} decls, stmt \end{array}} \tag{7.28}$$

Après avoir exécuté les instructions du corps de la méthode, la continuation revient dans la configuration où la tâche `INITIAL_INVITE` doit être exécutée. Les règles 7.29 et 7.30 décrivent respectivement l'exécution de la continuation selon que la méthode retourne une réponse avec un succès, respectivement une erreur. Dans le premier cas, seul l'état global est mis à jour avec le nouvel environnement et la nouvelle branche. En cas d'erreur, la persistance de la session est initialisée afin de clore la session. Dans ce cas, la persistance prend la valeur fausse et la session sera immédiatement détruite lors de l'appel à la fonction `end_session` réalisé par la fonction `continue_session`.

$$\frac{\begin{array}{c} \mathsf{lookup\_branches}(\sigma, address) = branches \\ \mathsf{continue\_session}(\phi, \sigma, address, \rho_{\mathsf{envs}}, \mathsf{add}(b, branches)) = \sigma' \end{array}}{\begin{array}{r} \langle \sigma, address \rangle, \rho \models_{\text{mc}} \langle \texttt{INITIAL\_INVITE}\ \ \phi \rangle, /\texttt{SUCCESS}/resp\,(resp\_headers)\,, b \\ \Rightarrow /\texttt{SUCCESS}/resp\,(resp\_headers)\,, \sigma' \end{array}} \tag{7.29}$$

$$\frac{\begin{array}{c} \mathsf{set\_persistence}(\sigma, address, \mathsf{false}) = \sigma' \\ \mathsf{lookup\_branches}(\sigma', address) = branches \\ \mathsf{continue\_session}(\phi, \sigma', address, \rho_{\mathsf{envs}}, \mathsf{add}(b, branches)) = \sigma'' \end{array}}{\begin{array}{r} \langle \sigma, address \rangle, \rho \models_{\text{mc}} \langle \texttt{INITIAL\_INVITE}\ \ \phi \rangle, /\texttt{ERROR}/resp\,(resp\_headers)\,, b \\ \Rightarrow /\texttt{ERROR}/resp\,(resp\_headers)\,, \sigma'' \end{array}} \tag{7.30}$$

Les invocations des méthodes intermédiaires et terminales sont similaires excepté le fait qu'elles n'invoquent pas la fonction `create_session`. À la fin d'une méthode intermédiaire, la session continue systématiquement et la règle de continuation est donc similaire à la règle en cas de succès (Règle 7.29). À la fin d'une méthode terminale, la session est systématiquement close, la règle de continuation est donc similaire à la règle en cas d'erreur (Règle 7.30). Il est à noter que la persistance est toujours fausse pour les méthodes initiales. Elle varie cependant en fonction de la méthode finale invoquée. Le troisième argument de la fonction `set_persistence` est ajusté en conséquence selon le tableau 7.4.

Expressions `forward` L'évaluation d'une expression `forward` provoque l'arrêt momentané du service SPL en cours d'exécution et le retour du contrôle vers la machine virtuelle SIP sous-jacente, qui relaye alors la requête sur le réseau avant de traiter les autres requêtes entrantes. Quand la réponse arrive, la machine virtuelle restaure le contexte du service ayant réalisé l'opération `forward`. Cette relation de co-routinage [HFW84] entre les services SPL et la machine virtuelle SIP nécessite que l'opération `forward` fournisse suffisamment de contexte à la machine virtuelle pour que l'exécution de la méthode puisse reprendre et finir son

Méthode	**Persistance**
méthodes initiales	non
`undeploy`	non
`unregister`	variable (en fonction de l'administrateur)
	oui en cas d'expiration de l'alarme
`uninvite`	non
`unsubscribe`	non

TAB. 7.4: Valeur de la persistance en fonction de la terminaison de session

exécution. Nous employons pour cela une stratégie standard de co-routinage avec passage de continuation.

La sémantique de l'opération `forward` est définie par les trois configurations suivantes :

- Expressions : $\langle \sigma, address \rangle, \langle envs, uri, local\_env \rangle, s \models_{e} exp$
- Continuation des expressions : $\langle \sigma, address \rangle, \langle envs, uri, local\_env \rangle \models_{ec} s, value$
- Appel de la machine virtuelle SIP : $\mathsf{forward}(value, bool, s, headers, \sigma)$
- Retour de la machine virtuelle SIP : $\mathsf{forward\_response}(value, s, \sigma)$

La règle 7.31 commence par évaluer l'éventuelle expression fournie en argument du `forward`, c'est-à-dire une URI pour réaliser une redirection. Une fois l'expression calculée, le service s'interrompt et donne le contrôle à la machine virtuelle (Règle 7.32). Si aucune redirection n'est demandée, le service s'interrompt immédiatement et rend le contrôle à la machine virtuelle (Règle 7.33). Lorsqu'une réponse est reçue par la machine virtuelle, cette dernière restaure le contexte et le service comme le décrit la règle 7.34. L'exécution reprend en dépilant la continuation. Dans le cas d'une redirection en parallèle, c.-à-d. `parallel forward`, le second argument de la fonction forward prend la valeur vrai.

$$\frac{}{\tau, \rho, s \models_{e} \texttt{forward}\ \ exp \Rightarrow \tau, \rho, \texttt{FORWARD\_EXP} :: s \models_{e} exp} \tag{7.31}$$

$$\frac{\begin{array}{c}\mathsf{update\_envs}(\sigma, address, \rho_{\mathsf{envs}}) = \sigma' \\ \rho_{\mathsf{msg\_info}} = \langle m, \langle rq, headers \rangle \rangle\end{array}}{\begin{array}{l}\langle \sigma, address \rangle, \rho \models_{ec} \texttt{FORWARD\_EXP} :: s, v \\ \Rightarrow \mathsf{forward}(v, \mathsf{false}, (\texttt{FORWARD}\ \ address\ \ \rho_{\mathsf{msg\_info}}\ \ \rho_{\mathsf{local\_env}}) :: s, headers, \sigma')\end{array}} \tag{7.32}$$

$$\frac{\begin{array}{c}\mathsf{update\_envs}(\sigma, address, \rho_{\mathsf{envs}}) = \sigma' \\ \rho_{\mathsf{msg\_info}} = \langle m, \langle rq, headers \rangle \rangle\end{array}}{\begin{array}{l}\langle \sigma, address \rangle, \rho, s \models_{e} \texttt{forward} \\ \Rightarrow \mathsf{forward}(rq, \mathsf{false}, (\texttt{FORWARD}\ \ address\ \ \rho_{\mathsf{msg\_info}}\ \ \rho_{\mathsf{local\_env}}) :: s, headers, \sigma')\end{array}} \tag{7.33}$$

$$\frac{\begin{array}{c}\mathsf{lookup\_envs}(\sigma, address) = envs \\ \tau = \langle \sigma, address \rangle \qquad \rho = \langle envs, msg\_info, local\_env \rangle\end{array}}{\begin{array}{r}\mathsf{forward\_response}(resp, (\texttt{FORWARD}\ \ address\ \ msg\_info\ \ local\_env) :: s, \sigma) \\ \Rightarrow \tau, \rho \models_{ec} s, resp\end{array}} \tag{7.34}$$

7.3 Bilan

Nous avons présenté dans ce chapitre un langage dédié, nommé SPL, pour spécifier des services de routage de messages entre des services de communications. Ce langage est statiquement typé et possède des abstractions spécifiques aux services de routage pour des communications fondées sur SIP. Nous retiendrons comme abstractions les sessions hiérarchiques et les opérations de signalisation. Ces abstractions explicitent le comportement d'un service et fournissent des opportunités pour optimiser la gestion de l'état automatiquement et vérifier la cohérence du service vis à vis du protocole de communications SIP sous-jacent. Nous avons présenté en détail deux analyses statiques : l'une vérifiant la cohérence de type dans un service et garantissant entre autres qu'une réponse unique est toujours retournée (Section 7.2.2.1), et l'autre vérifiant la bonne utilisation de l'opération de signalisation `forward` (Section 7.2.2.2). Nous avons également présenté la sémantique dynamique des constructions principales du langage (Section 7.2.3). Nous pouvons désormais, à l'aide de la sémantique de SPL, en réaliser une implémentation.

Chapitre 8

Mise en œuvre de SPL

Dans ce chapitre nous décrivons notre implémentation du langage SPL [BCL+06a, PCRL07] ainsi qu'un exemple de service SPL exploitant les fonctionnalités du langage. Nous précisons dans un premier temps le domaine de notre étude définissant les hypothèses de l'implémentation. Dans un second temps, nous décrivons le processus global de développement des services SPL. Enfin, la présentation de la chaîne de compilation détaille chaque phase de ce processus. Nous concluons par l'explication d'un service SPL ayant servi à la validation du processus et de son implémentation.

8.1 Domaine de l'étude

La mise en œuvre que nous présentons dans ce chapitre se limite à une seule pile protocolaire SIP : l'implémentation de référence de l'interface de programmation JAIN SIP. La couche logicielle permettant de faire le lien entre le protocole SIP et le langage est par conséquent fondée sur le langage Java. Cette couche logicielle constitue une machine virtuelle. Dans notre étude, nous avons développé un premier interprète, en O'Caml [Ler97], que nous avons expérimenté sur différentes plate-formes : Windows, Linux et Mac. Cet interprète fonctionne en co-routinage avec la machine virtuelle SIP. Une fois notre approche ainsi validée, nous avons fait évoluer cet interprète vers une implémentation Java afin d'obtenir une implémentation plus homogène.

Les différents éléments présentés dans ce chapitre ne sont cependant pas spécifiques aux langages Java et O'Caml et peuvent être implémentés à l'aide d'autres langages de programmation généralistes. Ainsi, il semble raisonnable de cibler des langages de programmation tels que C, C++ ou C#. Bien qu'une telle implémentation puisse s'avérer moins aisée en C, des gains de performance en temps d'exécution peuvent être escomptés.

8.2 Processus de développement

La figure 8.1 illustre le processus de développement d'un service de routage fondé sur SPL. Dans un premier temps, un service est développé par un programmeur en SPL. Une phase de vérifications est ensuite effectuée pour déceler d'éventuelles incohérences vis à vis des contraintes du protocole SIP et de la pile sous-jacente. Cette phase de vérifications terminée, un générateur de code est exécuté et produit une représentation adaptée à un interprète. L'interprétation des services est déclenchée par les événements que produit la machine virtuelle. Lors

de l'interprétation, les opérations de signalisation fournies par la machine virtuelle sont invoquées par l'interprète. La machine virtuelle et un interprète du langage SPL constituent un serveur d'applications pour les services de routage. Ce serveur d'applications offre une console d'administration permettant d'installer et de déployer des services pour les utilisateurs de la plate-forme de communications.

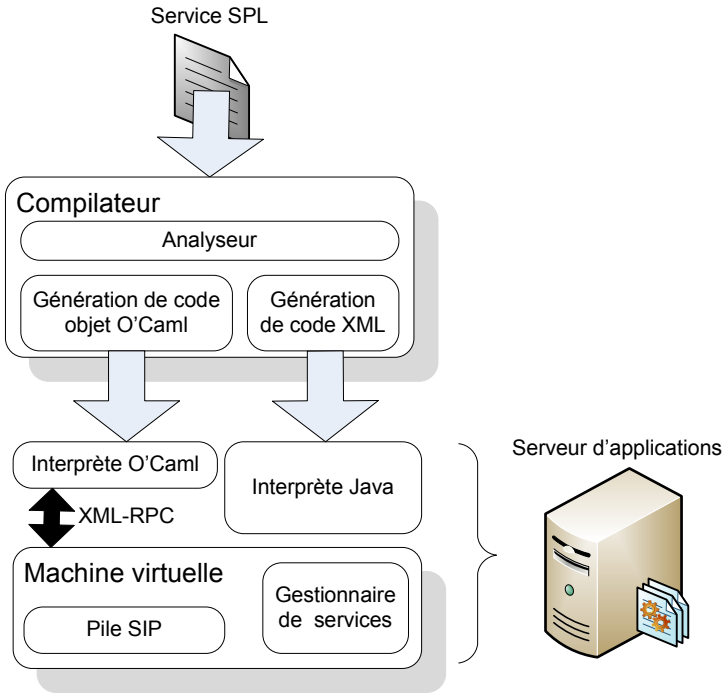


FIG. 8.1: Processus de développement de services de routage en SPL

8.3 Chaîne de compilation

La chaîne de compilation du langage SPL se compose de trois éléments. Tout d'abord un analyseur vérifie la validité des services vis à vis de propriétés pouvant être déterminées statiquement. Deux générateurs de code produisent ensuite au choix une représentation binaire ou une représentation XML du service. Celles-ci sont destinées respectivement à l'interprète O'Caml et l'interprète Java. Ces deux interprètes coopèrent avec la machine virtuelle SIP, dédiée à SPL. Cette machine virtuelle, associée à l'un des interprètes, forme un serveur d'applications SIP pour des services SPL.

8.3.1 Analyseur

L'analyseur de service SPL se décompose en plusieurs parties totalisant plus de 4 500 lignes de code O'Caml : d'une part un analyseur lexical et syntaxique standard, d'autre part des analyseurs spécifiques aux propriétés des services de routage reposant sur le protocole SIP. Ces

analyseurs spécifiques représentent environ 800 lignes de code dont plus de 600 pour l'analyse de type.

L'analyse de type permet de garantir de nombreuses propriétés du domaine. Elle permet de garantir, par exemple, la bonne utilisation des en-têtes des messages SIP ou qu'une réponse est toujours retournée suite à une requête. Elle garantit également qu'une redirection a toujours lieu vers une URI bien formée.

8.3.2 Générateurs de code et interprètes

L'environnement d'exécution des services SPL repose sur deux interprètes (O'Caml et Java). Ils sont tous deux issus d'une implémentation directe de la sémantique dynamique. Cette sémantique a en outre permis un développement rapide des interprètes en respectivement 1 et 2 semaines, comme l'indique le tableau 8.1.

Ces deux implémentations interprètent une représentation adaptée des services. L'implémentation O'Caml fonctionne grâce à une représentation sérialisée d'objets O'Caml tandis que l'implémentation Java importe une représentation XML sous forme d'objets Java. Deux générateurs de code permettent de réécrire les services validés par l'analyseur dans l'une ou l'autre des représentations.

	O'Caml	**Java**
Temps de développement	1 semaine	2 semaines
Nb. de lignes	2 175	5 786
Nb. de mots	10 855	17 040
Nb. de fonctions/méthodes	82	36
Nb. de modules/classes	6	45

TAB. 8.1: Comparaison des implémentations O'Caml et Java

8.3.3 Machine virtuelle

Chacune des implémentations de l'interprète coopère avec une machine virtuelle SIP dédiée à SPL (SIP VM)[BCL+06b]. La SIP VM a en charge de fournir, pour l'ensemble des services SPL, les abstractions de services et de sessions ainsi que la gestion de l'état. De plus, cette machine virtuelle doit offrir une architecture ouverte et flexible permettant aux services d'interagir avec des ressources extérieures via les modules SPL.

Afin de répondre à ces besoins, nous avons développé plusieurs composants logiciels : un gestionnaire de services, un modérateur de services, une couche d'abstraction du protocole, une couche d'abstraction des services et un environnement d'exécution ouvert. L'architecture de ces composants est représentée en Figure 8.2.

Gestionnaire de services Le gestionnaire de services fournit une interface pour contrôler le cycle de vie des services. Les services sont stockés dans un dépôt local sous leur forme exécutable précédemment générée. Ces opérations sont gérées à l'aide d'une console d'administration à distance. Une fois installés, les services peuvent être activés pour un utilisateur ou un groupe d'utilisateurs. Lors de l'activation d'un service, un événement de déploiement est

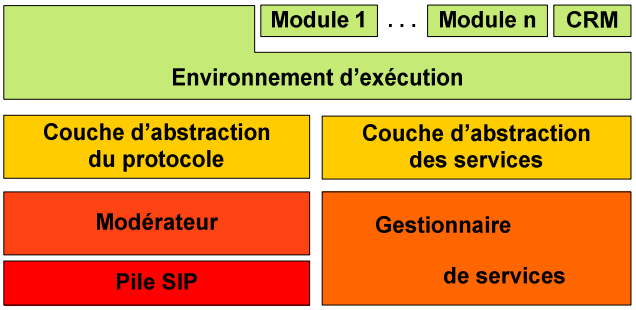


FIG. 8.2: Architecture du serveur d'applications

généré par le gestionnaire de services. Les utilisateurs ainsi que les groupes sont également administrés via la console d'administration.

Couche d'abstraction des services Lorsqu'un service est activé via le gestionnaire de services, un événement `deploy` est émis vers l'environnement d'exécution. La méthode correspondante du service est alors évaluée par l'interprète. De manière similaire, quand un service est désactivé, un événement `undeploy` est émis.

Bien que ces événements soient générés par la plate-forme et non pas issus d'un événement protocolaire SIP, l'environnement d'exécution permet de les traiter uniformément au niveau des services SPL. Enfin, cette couche d'abstraction est responsable de la gestion de l'état des services lors des phases de sauvegarde et de restauration qui ont lieu entre les événements.

Modérateur automatique Lorsqu'un message SIP est reçu par la pile protocolaire, celle-ci transmet le message au modérateur. Le modérateur analyse alors le message SIP et détermine l'ensemble des services devant être exécutés pour un utilisateur donné. Cet ensemble de services forme une liste ordonnée qui est réévaluée après chaque exécution d'un service. En effet, une redirection peut avoir eu lieu. Il n'est alors plus nécessaire d'exécuter les services restant pour cet utilisateur.

Couche d'abstraction du protocole Le rôle de cette couche est de réaliser le raffinement des événements protocolaires en événements SPL organisés selon le cycle de vie de la session à laquelle ils se rapportent. Les requêtes sont ainsi analysées pour déterminer s'il s'agit de la création d'une session, d'une opération intermédiaire ou de la terminaison d'une session. Le raffinement des événements opéré par cette couche d'abstraction prend également en compte les alarmes protocolaires ayant pu expirer.

Environnement d'exécution L'environnement d'exécution réagit aux événements produits par les couches d'abstraction. Il fournit ainsi aux services une vue haut niveau et unifiée des événements. L'environnement d'exécution organise de manière hiérarchique les sessions fournies par les deux couches d'abstraction sous-jacentes et facilite de fait la gestion de l'état au niveau des services.

Modules Les modules permettent d'étendre les fonctionnalités qu'offre le serveur d'application aux services SPL. Il s'agit d'une approche standard déjà exploitée dans ce but par d'autres

serveurs tels que Apache[Apa] ou OpenSer[Opea]. Dans le cas des services de communications, chaque service Internet est potentiellement une ressource et peut être exploité par un service de routage. Il est ainsi possible de développer des modules permettant d'accéder à des services locaux (par exemple, système de fichiers ou date et heure du système) ou des services distants (par exemple, services Web, bases de données, agendas partagés ou serveur de conférences).

Afin de valider notre approche, nous avons développé divers modules pour, par exemple, la gestion d'un journal, la manipulation d'un carnet d'adresses ou le contrôle d'une ressource multimédia (CRM). Le module CRM a entre autre été mis en œuvre lors du développement d'un service de file d'attente téléphonique.

8.4 Développement de services

Un service de file d'attente (Annexe C) a été développé afin de valider le langage et son implémentation. Ce service consiste en environ 200 lignes de SPL et permet la mise en relation avec un secrétariat. Une file d'attente est définie pour quatre personnes pendant qu'une cinquième est en relation avec le secrétariat. Les appels entrants sont tous routés vers un agent SIP fournissant des fonctionnalités multimédia. Cet agent est piloté grâce au module CRM et offre des fonctionnalités de pont audio, similaires à un serveur de conférence, et de lecture de messages pré-enregistrés.

Si une personne appelle le secrétariat et que celui-ci est disponible, les interlocuteurs sont mis en relation (premier cas en haut à gauche, Figure 8.3). Les personnes arrivant par la suite restent en attente. L'agent leur annonce en boucle leur rang dans la file (autres cas, Figure 8.3). Lorsque l'un des interlocuteurs raccroche, la personne suivante dans la file est mise en relation avec le secrétariat désormais disponible. Les autres personnes dans la file sont alors notifiées de leur nouveau rang. Si la file est pleine, les appels supplémentaires sont rejetés en indiquant de rappeler ultérieurement.

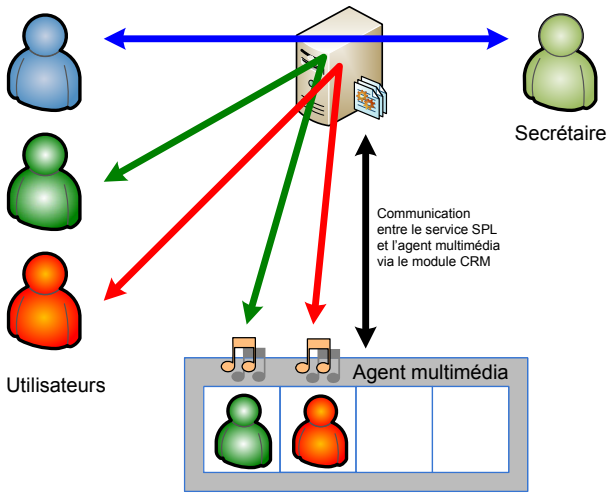


Fig. 8.3: Principe du service de file d'attente

Ce service met en œuvre plusieurs fonctionnalités du langage dont les sessions hiérarchiques, les opérations de signalisation et l'accès aux en-têtes. Il nécessite pas moins de 6

méthodes de la session `dialog` ainsi que plusieurs modules externes. Enfin, des fonctions internes ont également été mises en œuvre. La concision du programme et l'utilisation de l'analyseur ont permis de valider rapidement une version du service, sans erreur, avant son déploiement sur le serveur d'applications. La concision du programme permet enfin de facilement développer des nouveaux services de file d'attente avec plus de personnes ou des messages d'attente différents.

8.5 Bilan

Nous avons décrit dans ce chapitre le processus de développement de services de routage fondé sur l'approche SPL. Ce chapitre permet de se familiariser avec la chaîne de compilation. Il présente également l'architecture logicielle d'un serveur d'applications pour des services SPL. Les différents outils logiciels mis en œuvre durant le processus de développement ont été validés par l'exécution d'un service SPL d'environ 200 lignes gérant la signalisation d'un système de file d'attente et exploitant les diverses fonctionnalités du langage. Ce service relativement complexe en terme de fonctionnalités s'écrit simplement et de manière concise en SPL. Cette concision et l'utilisation de l'analyseur ont permis d'obtenir et valider rapidement le service. Sa concision permet aisément et rapidement de développer des variantes.

Chapitre 9

Pantaxou

Nous avons présenté dans les deux chapitres précédents le langage SPL, dédié au développement des services de routage. Nous allons maintenant décrire le langage Pantaxou, dédié au développement de services entités. Les services entités sont des entités communicantes. Contrairement aux services de routage qui ne font que traiter des communications en cours, les services entités peuvent également initier des communications vers d'autres entités communicantes.

Après un rappel des éléments constituant le domaine des services entités, nous définissons dans ce chapitre le langage Pantaxou. À l'instar du langage SPL, la définition du langage Pantaxou se compose d'une syntaxe, d'une sémantique statique et d'une sémantique dynamique.

9.1 Analyse du domaine

Nous appliquons la même démarche pour l'analyse de domaine que celle suivie lors de la conception du langage SPL. Après avoir précisé nos sources d'information, nous présentons dans cette section les points communs et les variations qui existent entre différents services entités. Nous complétons l'analyse de domaine par un cahier des charges pour le développement des services entités.

9.1.1 Sources d'information

L'analyse de domaine des services entités reprend les sources d'information exploitées lors de la conception du langage SPL telles que les services fondés sur les approches existantes et les RFC relatives à SIP. Nous profitons également de l'expérience acquise lors de la conception de SPL.

9.1.2 Points communs

Les services de communications que l'on considère sont des intervenants dans les communications ; nous parlerons d'entités communicantes ou de services entités. Pour classifier les communications entre les entités, nous définissons trois taxonomies : une pour les entités, une pour les modes de communications et une pour les types de données échangées.

Entités communicantes Dans le chapitre 2, nous avons vu quelques exemples de services entités tels que des caméras, des téléphones, des écrans et des capteurs de présence ou de

température. Tous les services avec une capacité de communication peuvent être considérés. Nous parlerons de fonctionnalité pour caractériser les capacités de communication d'un service. Il est facile d'imaginer une pléthore de services entités. Ces services sont caractérisés par deux attributs : des propriétés et des fonctionnalités. Les propriétés caractérisent les services et précisent, par exemple, la résolution d'un écran ou les codec installés sur un téléphone SIP. Les fonctionnalités expriment comment interagir avec les autres entités. Elles sont définies par deux éléments : le mode de communications mis en œuvre et le type des données échangées.

Modes de communications L'échange d'information entre des entités a lieu selon trois modes de communications : asynchrone, synchrone et flux.

Les communications asynchrones permettent au service qui initie la communication de ne pas rester bloqué. Il poursuit son exécution après sa requête auprès du service distant. Le service source souscrit à un type de données auprès d'un autre service. Ce dernier émet des données en direction des souscripteurs. Ce mode de communications est généralement appelé mode événementiel [EFGK03, Eug07].

Les communications synchrones impliquent que le service qui requiert une opération sur un service distant, se mette en attente d'une réponse. Cette réponse contient une information permettant au service source de poursuivre son exécution.

Le dernier mode de communications identifié est la communication par flux de données. Nous employons par la suite le terme de *session*[1]. Ce mode de communications est principalement utilisé pour le transport de la voix et de la vidéo mais il peut facilement être généralisé à d'autres types de données tels que des mesures physiques, par exemple l'humidité, la luminosité ou la température. Il est alors possible d'émettre ou de recevoir un flux de mesures de température pour, par exemple, asservir un climatiseur. Les sessions impliquent généralement une grande quantité de données. Afin de réduire ces quantités, des codec sont mis en place pour compresser les données. La mise en place d'une communication par session nécessite de convenir d'un codec entre les deux parties.

Types de données échangées Nous avons pu remarquer à plusieurs reprises qu'il existe une multitude de types de données pouvant faire l'objet d'échange entre des services. Parmi les types de données déjà introduits, il y a les données audio et vidéo, les mesures physiques, le texte pour ne citer que cela. Mais nous pourrions également parler de l'information de présence, de disponibilité, de localisation d'une personne ou d'un bien.

Il est nécessaire de décrire les données échangées pour définir un système de communications. Une donnée peut être décrite à l'aide de propriétés. Ces propriétés constituent elles-mêmes des données typées. Nous obtenons ainsi une description arborescente des types de données jusqu'à l'utilisation de types primitifs tels que des entiers ou des chaînes de caractères. Cette description est similaire à la définition Java d'un objet et de ses attributs. Par exemple, une image expose plusieurs propriétés comme une profondeur de couleur et une résolution. La profondeur de couleur peut être exprimée par un entier. La résolution est quant à elle décrite par un couple d'entiers précisant la hauteur et la largeur de l'image.

[1] Il est à noter que la notion de session Pantaxou n'est pas la même que celle en SPL.

9.1.3 Variations

Après avoir déterminé les points communs des services entités et de leurs communications, nous allons voir les variations entre les services, les modes de communications et les types de données.

Entités communicantes L'apparition de nouveaux périphériques nécessite une approche flexible, capable de supporter ces nouveaux services. Par exemple, une caméra équipée d'un détecteur de mouvement doit pouvoir être intégrée dans un environnement existant où d'autres caméras sont déployées. La nouvelle caméra fournissant au moins les mêmes fonctionnalités que les anciennes doit être observée de façon indifférenciée par les autres services. Les services existants, communiquant avec des caméras, ne sont pas affectés par les évolutions. Ce besoin de flexibilité suggère de définir les nouveaux périphériques en terme de périphériques existants auxquels sont ajoutés les nouvelles fonctionnalités. La définition d'un nouveau service est ainsi fondée sur la dérivation d'un ancien par l'ajout de propriétés et de fonctionnalités.

Une deuxième variation entre les services réside dans leurs fonctionnalités et les propriétés qui les caractérisent. Les services fournissent des fonctionnalités différentes et ils exposent donc une variété de combinaisons, de modes de communications et de types de données. Par exemple, un téléphone fournit une fonctionnalité de communication par session audio, tandis qu'une caméra fournit une fonctionnalité de session vidéo. Un capteur de mouvement émet des événements de détection. Enfin, un capteur de température est interrogé via une opération synchrone renvoyant la température courante. Concernant les propriétés des services, chaque service possède des propriétés spécifiques. Lors de la définition d'un service, il faut également définir l'ensemble des propriétés qui le caractérisent. Par exemple, un périphérique matériel a un emplacement physique. Un capteur de mouvement surveille les mouvements dans une zone délimitée. Une caméra ou un téléphone ne peuvent communiquer qu'à l'aide de certains codec.

Afin de faire face à l'abondance de services, un expert crée un modèle de l'environnement qu'il expertise. L'expert d'environnement définit les services présents et prévus dans l'environnement en fonction de scénarios d'usage. Selon l'environnement (par exemple, une école, une entreprise, une administration) dans lequel Pantaxou sera mis en œuvre, l'ensemble des services considérés pourra ainsi varier. Conceptuellement, un modèle d'environnement est associé à un environnement. En pratique, une réutilisation des parties communes entre deux modèles doit être mise en pratique afin de minimiser le coût de la définition d'un modèle. Cette réutilisation est de la responsabilité des experts définissant les modèles. Nous pouvons envisager la constitution d'une base de connaissance dans ce but.

Modes de communications Le plan de contrôle d'une communication est indépendant du plan de données. Par exemple, un service de surveillance connecte une caméra à un écran. Il initie les communications et les contrôle mais n'a pas de données à transmettre. Il reçoit une session vidéo de la caméra et la retransmet à un écran. Lors de la définition des fonctionnalités d'un service entité, il est par conséquent important de bien caractériser l'orientation du flux de données pour les communications de type session. Le flux peut ainsi être unidirectionnel comme par exemple pour un écran ou une caméra où la session vidéo est respectivement reçue et émise. Le flux peut également être bidirectionnel comme par exemple les sessions audio entre deux téléphones. L'orientation des données pour les deux autres modes de communications est implicite.

La signature d'une opération synchrone définit les échanges de données entre les deux interlocuteurs. Les arguments de l'opération indiquent les données émises, tandis que le type de retour indique les données reçues. De manière similaire, les opérations asynchrones que nous considérons sont fondées sur deux phases : une phase de souscription à des événements et une phase de réception des événements. Lors de la souscription, le souscripteur précise les événements qu'il souhaite recevoir ultérieurement. Dans la seconde phase, l'événement est porteur d'une donnée à destination des souscripteurs.

Types de données échangées Il existe de très nombreux types de données échangeables. Certaines propriétés sont par ailleurs communes entre plusieurs types. Par exemple, le type de la résolution d'une image définit à la fois la taille d'une image fixe, telle qu'une photo, ou la résolution d'une vidéo. Afin de favoriser la réutilisation des définitions de types, la définition des types de données doit être hiérarchique.

9.1.4 Cahier des charges

Les objectifs du langage Pantaxou sont similaires à ceux du langage SPL hormis la cible des services. Le langage Pantaxou doit faciliter le développement de services entités en offrant des abstractions favorisant la concision des programmes. Les programmes produits sont plus simples à comprendre et à maintenir lors d'évolutions ultérieures. De plus, l'utilisation d'une approche langage dédié est de nature à favoriser la réutilisation. L'environnement d'exécution et la chaîne de compilation permettent une réutilisation systématique de code et d'expertise pour chaque développement d'un nouveau service. Finalement, Pantaxou doit offrir des opportunités de vérifications permettant de garantir le bon fonctionnement global du système, si possible avant l'exécution. Des vérifications statiques doivent donc être introduites pour garantir une coordination valide des entités communicantes.

9.2 Définition du langage

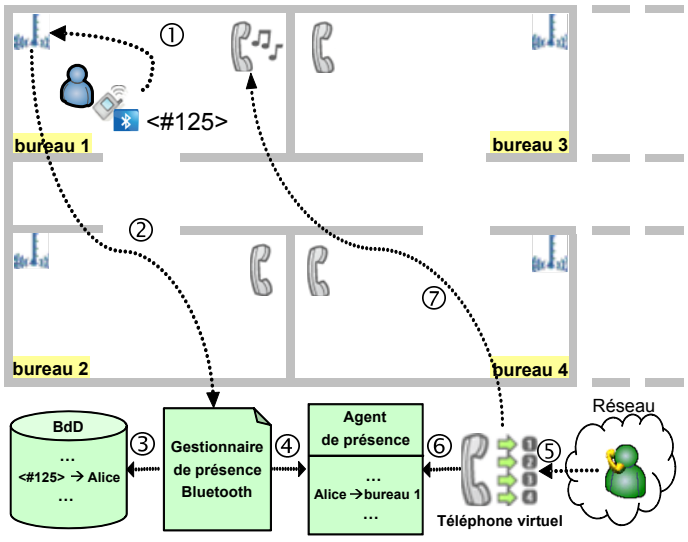
À la suite de l'analyse du domaine des services entités, nous définissons l'approche Pantaxou, sa syntaxe d'une part et sa sémantique statique et dynamique d'autre part.

9.2.1 Syntaxe

L'approche Pantaxou est une approche en deux temps. Le langage reflète cette décomposition et est composé de deux grammaires distinctes et complémentaires. La première grammaire permet de définir un modèle d'environnement. Nous parlerons du langage *PantaEnv*. Il comprend la définition abstraite des services entités de l'environnement et les types de données associés. La seconde grammaire exprime des services entités concrets en définissant leur logique. Nous parlons alors du langage *PantaLog*.

L'approche Pantaxou est décrite avec l'exemple d'une redirection des appels d'une secrétaire de son bureau vers celui où elle se trouve au moment de l'appel ; ce scénario est détaillé en Figure 9.1. L'objectif est de lui permettre de ne pas perdre d'appels, y compris pendant ses déplacements dans l'entreprise. Ce scénario nécessite de localiser la secrétaire dans l'enceinte du bâtiment de l'entreprise. Il met en œuvre divers services : des détecteurs de présence, un gestionnaire de présence, une base de données, un agent de présence et des téléphones dont un téléphone virtuel. Nous proposons d'exploiter la technologie Bluetooth pour résoudre le

problème de détection de la présence. Des équipements Bluetooth sont déployés de manière permanente dans les bureaux (par exemple, ordinateur fixe, clé USB Bluetooth) et détectent la présence de la secrétaire à l'aide d'un périphérique Bluetooth qu'elle porte (par exemple, son téléphone cellulaire personnel). Lorsqu'un lecteur Bluetooth détecte l'entrée d'un périphérique Bluetooth, il émet un événement à un gestionnaire de présence Bluetooth. Ce gestionnaire recherche l'identité du propriétaire du périphérique dans une base de données. Il émet à son tour un événement signalant la présence d'une personne dans un bureau particulier. L'agent de présence reçoit cet événement et stocke l'association entre la localisation et la personne (par exemple, la secrétaire). Si l'on cherche à joindre la secrétaire, un téléphone virtuel reçoit l'appel et consulte l'emplacement de la secrétaire dans la liste d'association précédemment remplie. L'appel est alors redirigé vers l'emplacement courant de la secrétaire.



1. Lorsqu'Alice entre dans le bureau 1, le lecteur Bluetooth détecte la présence de son périphérique Bluetooth.
2. Le lecteur Bluetooth publie un événement pour signaler la présence du périphérique ayant l'adresse ⟨#125⟩. Cet événement est reçu par le gestionnaire de présence Bluetooth.
3. Le gestionnaire recherche dans la base de données l'identité du propriétaire du périphérique ayant l'adresse ⟨#125⟩.
4. Le gestionnaire publie un événement signalant que Alice se trouve actuellement dans le bureau 1. Cet événement est reçu par l'agent de présence.
5. Lorsqu'un appel est à destination du secrétariat, une session audio arrive sur le service de téléphone virtuel gérant les appels.
6. Le téléphone virtuel demande à l'agent de présence où se trouve Alice.
7. Le téléphone virtuel transfert l'appel au téléphone du bureau 1.

FIG. 9.1: Scénario de redirection en fonction de la localisation

9.2.1.1 Définition d'un modèle d'environnement

Avant que des logiques de services entités soient développées et déployées, un expert définit un modèle de l'environnement. Ce modèle définit de façon abstraite les services qui doivent être développés. L'expert doit recenser et définir l'ensemble des types de données nécessaires aux scénarios d'usage qui se dérouleront dans l'environnement. L'expert définit ensuite les communications entre services et l'ensemble des services entités qui composent les différents scénarios. Les services sont définis par leur propriétés et leurs fonctionnalités. Les fonctionnalités détaillent les communications qui ont lieu entre les services. Les types de données, les communications et les services d'un modèle d'environnement sont regroupés dans un unique fichier (Figures 9.2, 9.3 et 9.4) écrit en PantaEnv.

Types de données échangées Les types de données sont définis de manière arborescente, en fonction de propriétés typées. Le langage PantaEnv définit des types primitifs tels que les entiers, les chaînes de caractères et les énumérations, pour typer les propriétés simples. Afin de favoriser la réutilisation des définitions des types, les types peuvent être définis hiérarchiquement. Il est ainsi possible de raffiner la définition d'un type ultérieurement sans impacter l'existant. Enfin, il appartient à l'expert de l'environnement de faire des choix de conception quant à la hiérarchie de types qu'il définit. Par exemple, une image peut être représentée de façon générique par un unique type de données ou un type de données peut être défini pour chaque type de format d'image devant être supporté par l'environnement. Ce type de choix de conception dépend fortement de l'environnement, et il appartient à l'expert de faire les choix les plus adaptés à son environnement. L'expert doit cependant tenir compte des évolutions que devra supporter l'environnement qu'il est en train de concevoir.

La figure 9.2 définit les déclarations des types de données nécessaires pour le scénario de redirection d'appels vers la secrétaire en déplacement. L'utilisation des sessions audio a été intégrée de façon native dans le langage. Le type `Audio` est un type fourni (ligne 1). Il doit être importé lorsque l'on souhaite l'exploiter dans un environnement particulier pour intégrer des périphériques SIP natifs, comme des téléphones SIP. Ces périphériques sont manipulés comme des services Pantaxou, bien qu'ils n'aient pas été développés selon cette approche. Nous trouvons ensuite la définition d'un bureau, modélisée par un identifiant de type entier naturel. Le type `Room` (ligne 3) sert à définir l'information de présence d'une personne. L'URI et le statut d'une personne sont également associés à la présence (lignes 7 à 11). Enfin, les types `DbBluetoothInfo` et `BluetoothDetection` servent respectivement lors de la consultation de la base de données et lors de la détection d'un périphérique Bluetooth (lignes 13 à 20).

```
1  import Audio;
2
3  struct Room {
4    int number;
5  }
6
7  struct Presence {
8    uri entity;
9    bool status;
10   Room room;
11 }
12
13 struct DbBluetoothInfo {
14   uri ownerName;
15 }
16
17 struct BluetoothDetection {
18   string bluetoothAddress;
19   int signalStrengh;
20 }
21
```

FIG. 9.2: Déclaration des types de données du modèle d'environnement

Modes de communications Nous définissons trois modes de communications pour les services Pantaxou : les commandes, les événements et les sessions. Ils correspondent respectivement aux communications synchrones, asynchrones et flux de données.

Les commandes sont définies de manière très similaire à des interfaces Java. Une commande est une liste nommée de signatures de méthodes. Ces signatures sont typées avec des types de donnée primitifs ou définis par l'expert d'environnement. La figure 9.3 donne la définition des commandes nécessaires pour notre scénario. Elles permettent respectivement de consulter la base de données et la liste d'association.

```
22 command DbQueryBluetooth {
23   DbBluetoothInfo getInfoBT(string bluetoothAddress);
24 }
25
26 command GetRoom {
27   Room getRoom(uri user);
28 }
29
```

FIG. 9.3: Déclaration des commandes du modèle d'environnement

Les événements et les sessions ne nécessitent pas de définition par l'expert de l'environnement. Il peut directement manipuler ces abstractions lors de la définition des services du modèle d'environnement (Figure 9.4). Les mots-clés `event` et `session` sont prévus à cet effet. Une notation similaire aux types paramétrés de Java 1.5 permet d'associer un mode de communication et un type de données précédemment défini. Finalement, le mot-clé `command` permet d'associer des commandes précédemment déclarées à des services.

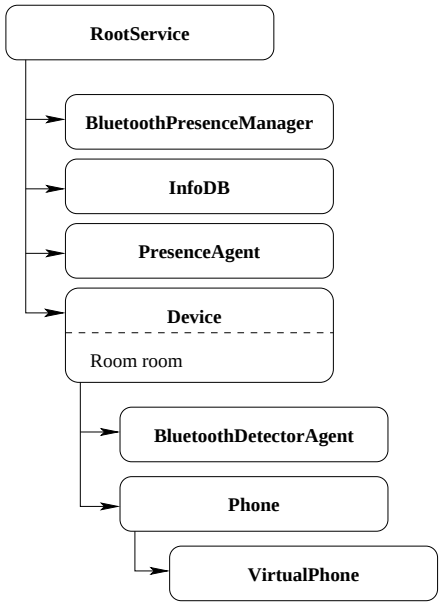
L'orientation du flux de données pour les communications de type session est défini par un modificateur. Il en existe trois, ' `!`', ' `?`', '='. Ils définissent respectivement les flux émis, reçus et bidirectionnels. La définition d'un téléphone indique par exemple que ce type de services émet et reçoit des sessions audio bidirectionnelles (lignes 54 et 55).

Taxonomie des services Pour faciliter la déclaration de ces services et la réutilisation, nous proposons une organisation hiérarchique des services (Figure 9.4-(b)). Ainsi, les capteurs Bluetooth (`BluetoothDetectorAgent`) et les téléphones (`Phone` et `VirtualPhone`) sont regroupés sous un nœud unique périphérique (`Device`). La figure 9.4-(a) donne la liste des services présents dans l'environnement pour réaliser notre scénario. Le mot-clé `extends` permet de définir une relation d'héritage entre les services. Les déclarations de services héritent à la fois des propriétés des services parents et de leurs fonctionnalités. Le téléphone virtuel se comporte ainsi comme un téléphone standard.

La déclaration des services permet de déclarer le couplage existant entre les services, au niveau de l'environnement. Pour ce faire, les mots-clés `requires` et `provides` déclarent les fonctionnalités respectivement requises et fournies par un service. La portée d'une fonctionnalité, vis à vis de la classe de services distants, est également précisée via les mots-clés `from` et `to`. Chaque déclaration d'une fonctionnalité permet ainsi un couplage fort entre deux types de service. Par exemple, le service `VirtualPhone` nécessite la commande `GetRoom` d'un service `PresenceAgent` (ligne 58). Le dual de cette déclaration consiste à déclarer le service

```
30 service RootService {}
31 service BluetoothPresenceManager extends RootService {
32   requires event<BluetoothDetection>
33     from BluetoothDetectorAgent;
34   requires command<DbQueryBluetooth> from InfoDB;
35   provides event<Presence> to PresenceAgent;
36 }
37 service InfoDB extends RootService {
38   provides command<DbQueryBluetooth>
39     to BluetoothPresenceManager;
40 }
41 service PresenceAgent extends RootService {
42   requires event<Presence>
43     from BluetoothPresenceManager;
44   provides command<GetRoom> to RootService;
45 }
46 service Device extends RootService {
47   Room room;
48 }
49 service BluetoothDetectorAgent extends Device {
50   provides event<BluetoothDetection>
51     to BluetoothPresenceManager;
52 }
53 service Phone extends Device {
54   provides session<=Audio> to Phone;
55   requires session<=Audio> from Phone;
56 }
57 service VirtualPhone extends Phone {
58   requires command<GetRoom> from PresenceAgent;
59 }
```

(a) Déclarations des services



(b) Hiérarchie des services

FIG. 9.4: Services d'un modèle d'environnement

`PresenceAgent` comme fournissant la commande `GetRoom` à l'ensemble des services, et *a fortiori*, par héritage au service `VirtualPhone` (ligne 44). Une propriété `room` de type `Room` (ligne 47) est déclarée pour les services `Device`. Elle permet de localiser n'importe quel périphérique.

9.2.1.2 Définition d'une logique de coordination de services

Lorsqu'un expert a modélisé un environnement, des développeurs peuvent ensuite développer des services entités. Ces services sont des instances des services modélisés dans l'environnement. Ils doivent à ce titre respecter certaines contraintes. Nous définissons la grammaire du langage de logiques Pantaxou, PantaLog, pour le développement de services entités. L'objectif de ce langage est de définir des logiques implémentant des services abstraits de l'environnement. Ces logiques des services entités permettent la coordination des services d'un environnement particulier de manière abstraite. En effet, les logiques ne font référence à aucune instance de service de l'environnement. Elles s'appuient sur les déclarations abstraites des services et sur un système de découverte de services reposant sur les propriétés des services.

Les figures 9.5, 9.6 et 9.7 présentent les services nécessaires au scénario de la secrétaire. Dans ce scénario, le gestionnaire de présence Blutooth, c.-à-d. le service `BluetoothPresenceManager` (Figure 9.5), coordonne l'interaction entre les détecteurs Bluetooth disséminés dans le bâtiment et la base de données. Le gestionnaire émet ensuite des événe-

ments `Presence`. Ces événements sont reçus par l'agent de présence (Figure 9.6). L'agent de présence maintient l'association entre les personnels de l'entreprise et leur localisation. Il peut être consulté pour déterminer le bureau actuel d'une personne donnée. Le téléphone virtuel représente le secrétariat (Figure 9.7) et redirige les appels vers les téléphones dans les bureaux. Il consulte pour cela l'agent de présence afin de déterminer comment router les appels.

La définition d'un service commence par l'importation des déclarations (Figure 9.5, lignes 1 à 7) du modèle nécessaires pour le service. Le développeur définit ensuite le nom du service à l'aide d'une URI en précisant quel service du modèle il implémente (ligne 9). Le point d'entrée du service se trouve au niveau du bloc de code `initial` (ligne 31). Ce bloc est exécuté lors de l'activation du service dans l'environnement. Il est également possible de déclarer un bloc `final` qui sera exécuté lors de la désactivation. Nous trouvons dans le corps du service la définition des services nécessaires au bon fonctionnement de la logique. Ces définitions servent lors de la découverte de services. Elles déterminent les propriétés particulières que doivent avoir les services d'un type particulier. Le gestionnaire de présence doit par exemple recevoir les événements de détection des identifiants Bluetooth provenant de l'ensemble des détecteurs. Le gestionnaire doit également consulter une base de données. Ces deux types de service sont définis par l'instance du gestionnaire de présence (lignes 11 et 12). Le domaine précise que le gestionnaire nécessite des événements `BluetoothDetection` provenant de détecteur Bluetooth. Le service déclare l'événement `btDetectionEvt` de type `BluetoothDetection` provenant des `bluetoothDetectorAgent`. La notation `[*]` précise que la souscription de l'événement sera fait auprès de tous les services correspondant à la définition d'un `bluetoothDetectorAgent` (ligne 14 à 16). La déclaration suivante permet de localiser une base de données grâce à l'opérateur `new` (ligne 18). Cet opérateur exécute une découverte de services. La découverte est fondée sur la déclaration de service précédente de la base de données (lignes 12). Le développeur déclare ensuite le bloc de code, identifié par `btEvtReception`, traitant les événements reçus de type `btDetectionEvt` (lignes 20 à 29). Lorsqu'un événement `e` est reçu, la base de données est consultée afin de déterminer le propriétaire du périphérique ayant l'adresse Bluetooth détectée (ligne 22). Un nouvel événement `p` de type `Presence` est initialisé grâce au numéro identifiant le bureau issu du détecteur Bluetooth et l'URI identifiant un employé issue de la base de données (lignes 23 à 27). Cet événement est finalement publié dans l'environnement vers des éventuels souscripteurs (ligne 28). Afin de recevoir des événements de détection Bluetooth (ligne 14), le service souscrit aux événements `btDetectionEvt` (ligne 20) provenant des `bluetoothDetectorAgent` (ligne 11) en adoptant le comportement `btEvtReception` (ligne 31). L'exécution de l'opération `adopt` réalise trois tâches. Premièrement, les services de type `BluetoothDectectionAgent` présent dans l'environnement sont découverts. Deuxièmement, le service courant souscrit à l'événement de type `BluetoothDetection` auprès de chaque service découvert. Enfin, le traitement `btEvtReception` est enregistré auprès de l'environnement d'exécution. Il sera exécuté à chaque réception d'un événement.

L'agent de présence (Figure 9.6) souscrit à l'événement publié par le gestionnaire de présence que nous venons de décrire (Figure 9.5). L'agent doit implémenter la commande `GetRoom` qui contient une unique méthode `getRoom`. L'implémentation de cette méthode retourne la dernière pièce où a été localisé un employé particulier, donné en argument (lignes 22 à 24). Pour ce faire, la liste d'associations est implémentée à l'aide d'une table de hachage. Cette table est maintenue à jour par le traitement associé à la réception des événements

```
1  import datatype BluetoothDetection;
2  import datatype DbBluetoothInfo;
3  import datatype Presence;
4  import datatype Room;
5  import command DbQueryBluetooth;
6  import service Service.InfoDB;
7  import service Service.BlueToothPresenceManager;
8
9  'sip:bt.manager@enseirb.fr' instantiates Service.BluetoothPresenceManager {
10
11   service<BluetoothDetectorAgent> bluetoothDetectorAgent;
12   service<InfoDB> database;
13
14   event<BluetoothDetection> from bluetoothDetectorAgent[*] {
15     signalStrengh > 50;
16   } btDetectionEvt;
17
18   database db = new database[1];
19
20   onReceive btEvtReception(btDetectionEvt e)
21   {
22     DbBluetoothInfo i = db.getInfoBT(e.data.bluetoothAddress);
23     event<Presence> {
24       entity = i.ownerName;
25       room = e.src.room;
26       status = true;
27     } p;
28     publish(new p);
29   }
30
31   initial { adopt(btEvtReception); }
32 }
```

FIG. 9.5: Gestionnaire de la présence Bluetooth

provenant du gestionnaire de présence (lignes 13 à 20).

L'agent de présence et le gestionnaire de présence peuvent servir dans d'autres scénarios où la localisation de périphériques ou d'employé est nécessaire et ne sont pas spécifiques au secrétariat. Le dernier service, le téléphone virtuel, est propre à notre scénario et réalise la redirection des appels du secrétariat vers le bureau où se trouve la secrétaire.

Le service de redirection du secrétariat (Figure 9.7) définit une instance d'un téléphone virtuel tel que déclaré dans le modèle d'environnement. Il détermine la localisation de la secrétaire et définit pour cela le service `PresenceAgent` (ligne 10). Il récupère une référence pour communiquer lors d'une opération de découverte de services (ligne 14). Le service définit également un service générique de type `Phone` afin de pouvoir traiter les appels entrants (lignes 9 et 12). Le corps du service se trouve dans le bloc de code traitant les appels entrants (lignes 16 à 31) ; ce bloc est activé avec le mot-clé `adopt` (ligne 33), comme pour les événements. Lorsqu'une session audio `s` arrive, c.-à-d. lors d'un appel entrant, un service `myPhone` de type `Phone` dans le bureau où se trouve la secrétaire est déclaré (lignes 19 à 21). L'agent de présence est consulté à ce titre (ligne 17) pour déterminer la pièce courante de la secrétaire. Un unique service `myPhone` est localisé lors de l'opération de découverte de services (ligne 22) et la session entrante `s` est redirigée vers ce dernier qui se trouve dans la même pièce que la secrétaire. Si l'appel est accepté, le service se met en attente d'un nouvel appel. Nous pourrions alors changer l'état de la secrétaire et préciser qu'elle se trouve actuellement au

```
1  import datatype Room;
2  import datatype Presence;
3  import service Service.BluetoothPresenceManager;
4
5  'sip:pres.agent@enseirb.fr' instantiates Service.PresenceAgent {
6
7    service<BluetoothPresenceManager> bm;
8
9    event<Presence> from bm[*] presenceEvt;
10
11   HashTable htable;
12
13   onReceive presenceReception(presenceEvt e) {
14     Presence p = e.data;
15     if (p.status) {
16       htable.put(p.entity, p.room);
17     } else {
18       htable.remove(p.entity);
19     }
20   }
21
22   Room getRoom(uri user) {
23     return htable.get(user);
24   }
25
26   initial {
27     adopt(presenceReception);
28     htable = new HashTable<uri, Room>();
29   }
30 }
```

FIG. 9.6: Agent de présence

téléphone. Si l'appel n'aboutit pas, l'erreur est retournée à l'appelant et le service se met en attente d'un nouvel appel. Nous pourrons dans ce cas, réaliser une nouvelle redirection vers une boîte vocale par exemple.

9.2.2 Sémantique statique

La décomposition du développement des services Pantaxou en deux temps (modélisation d'un environnement et implémentation de services) permet des vérifications complémentaires. La première phase consiste à vérifier la cohérence du modèle. Une seconde phase permet de vérifier la cohérence d'une implémentation d'un service vis à vis de la déclarations de ce service dans le modèle.

9.2.2.1 Analyses au niveau du modèle d'environnement

L'utilisation et la définition des types de données, des fonctionnalités et des services sont contrôlées. Le couplage fort entre les services d'un modèle permet de vérifier que le dual d'un service est bien défini. C'est-à-dire que nous pouvons garantir que pour un service donné, l'ensemble des fonctionnalités qu'il fournit sont bien utilisées par au moins un service du modèle et que l'ensemble des fonctionnalités qu'il requiert sont bien disponibles et fournies par au moins un service du modèle.

```
1  import datatype Audio;
2  import datatype Room;
3  import command GetRoom;
4  import service Service.PresenceAgent;
5  import service Service.Device.Phone;
6
7  'sip:secretary@example.com' instantiates Service.Device.Phone.VirtualPhone {
8
9    service<Phone> phone;
10   service<PresenceAgent> pa;
11
12   session<Audio> from phone[*] inSess;
13
14   pa agent = new pa[1];
15
16   onReceive callReception(inSess s) {
17     Room r = agent.getRoom('sip:alice@example.com');
18     if (r == null) r = 42;
19     service<Phone> {
20       room = r;
21     } myPhone;
22     invite(new myPhone[1], s) {
23       accepted(activeSession<Audio> aas) {
24         // An active audio session (aas)
25         // is accessible.
26       }
27       rejected() {
28         reject(s); // Forward the reject.
29       }
30     }
31   }
32
33   initial { adopt(callReception); }
34 }
```

FIG. 9.7: Service de redirection d'appels (Téléphone virtuel)

Pour cette analyse, l'environnement e_s associe le nom d'un type de service à l'ensemble des fonctionnalités qui le caractérise. Chaque fonctionnalité est décrite par le triplet (d, m, id) où d est la direction (c.-à-d. Provides ou Requires), m le mode de communication (c.-à-d. command, event ou session), et id est le nom du type de service distant. Nous définissons l'opérateur relationnel binaire $\sqsubseteq$ pour dénoter une relation de sous-typage entre les services. Nous écrirons ainsi $A \sqsubseteq B$ pour signifier que A est un sous-type de B, quand A et B sont tous deux, soit des types de services, soit des types de données. Pour les commandes, $A \sqsubseteq B$ si et seulement si *(ssi)* A et B sont la même commande, et pour les événements ou les sessions, $A \sqsubseteq B$ *ssi* le type de données associé à A est un sous-type du type de donnée associé à B.

La propriété (9.1) garantit que chaque fonctionnalité fournie est requise par au moins un type de service. Cette propriété garantit la cohérence du modèle en terme de connexion entre les services. Par exemple, dans un modèle bien formé, il ne peut y avoir de perte d'événement. C'est une propriété essentielle pour des événements critiques tels qu'une alarme incendie. Concernant les fonctionnalités de type commande, cette propriété évite du code mort.

$$\begin{aligned} &\forall id_s \in \mathsf{dom}(e_s),\ \forall(\mathsf{Provides}, m, id_t) \in e_s(id_s), \\ &\quad \exists id_{t'} \in \mathsf{dom}(e_s),\ \exists(\mathsf{Requires}, m', id_{s'}) \in e_s(id_{t'}), \\ &\qquad m \sqsubseteq m' \wedge id_{t'} \sqsubseteq id_t \wedge id_{s'} \sqsubseteq id_s \end{aligned} \tag{9.1}$$

La propriété (9.2) permet de vérifier que chaque fonctionnalité requise est fournie par au moins un type de service. Cette propriété garantit que chaque fonctionnalité utilisée est bien définie.

$$\begin{array}{c} \forall id_s \in \mathsf{dom}(e_s),\ \forall(\mathsf{Requires}, m, id_t) \in e_s(id_s), \\ \exists id_{t'} \in \mathsf{dom}(e_s),\ \exists(\mathsf{Provides}, m', id_{s'}) \in e_s(id_{t'}), \\ m' \sqsubseteq m \wedge id_t \sqsubseteq id_{t'} \wedge id_s \sqsubseteq id_{s'} \end{array} \tag{9.2}$$

Le langage PantaEnv définit l'orientation des flux de données des sessions avec les modificateurs ' ! ', ' ? ' et '='. La cohérence de l'orientation des flux est vérifiée de manière similaire. Nous autorisons la connexion d'un flux bidirectionnel avec un flux unidirectionnel. Il est ainsi possible d'utiliser, par exemple, un visiophone pour consulter une caméra.

9.2.2.2 Analyses au niveau de la logique des services

Lorsqu'un modèle est valide, son environnement d'exécution dédié peut être généré. Afin de déployer du code utilisateur dans cet environnement, les développeurs programment des services en PantaLog. Ces services sont vérifiés par le compilateur PantaGen. Le compilateur vérifie que les fonctionnalités mises en œuvre par les services correspondent bien à celles déclarées dans le modèle PantaEnv et uniquement celles-ci.

Nous définissons les règles permettant la vérification des fonctionnalités de type commande, événement et session. Dans ces règles, Γ représente l'environnement et est composé des types de service Σ, des commandes Ξ et des types de données Δ. On note Ξ^{-1} la fonction qui associe à une signature de méthode la commande à laquelle elle appartient. μ est l'ensemble des fonctionnalités définies pour le type de service qui est vérifié, et τ représente l'association des variables du service à leur type. Les règles des instructions font également référence au type t qui est attendu en retour du bloc de code où l'instruction apparaît. Les instructions ont le type Void à l'exception de l'instruction `return`.

Services La règle (9.3) définit la sémantique d'une déclaration de service. La description du type de service est tout d'abord recherchée dans le modèle PantaEnv à partir du nom du type de service. Les propriétés *properties*, déclarées par l'implémentation, sont ensuite vérifiées vis à vis de celles déclarées dans le modèle, τ_{prop}. Finalement l'environnement du service est retourné avec une nouvelle association entre l'identifiant *id* et le type du service.

$$\frac{\Gamma \vdash_{\text{spa}} \mathit{servicePath} : t \qquad t = \mathbf{ServiceType}(\_, \_, \tau_{\mathsf{prop}}) \qquad \Gamma,\ \mu,\ \tau, \tau_{\mathsf{prop}} \vdash_{\text{p}} \mathit{properties}}{\Gamma,\ \mu,\ \tau \vdash_{\text{d}} \texttt{service<}\mathit{servicePath}\texttt{>}\ \ \{\mathit{properties}\}\ \ \mathit{id}\texttt{;} : \tau[\mathit{id} \mapsto t]} \tag{9.3}$$

Les règles (9.4) et (9.5) permettent d'instancier les déclarations de service. Elles prennent une déclaration de service de type **ServiceType** et produisent un type ServiceInstType. Ce type correspond a des services qui ont été préalablement découvert dans le système grâce à l'opérateur `new`. La règle (9.4) s'applique lorsqu'un nombre particulier d'entités est requis lors de la découverte. La règle (9.5) s'applique lorsque toutes les entités doivent être découvertes.

$$\frac{\tau(service) = \textbf{ServiceType}(\_,\_,\_) \quad i \in \mathbb{N}^* \quad \mathsf{ServiceInstType}(i, \tau(service)) = t}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} \texttt{new}\ \ service\texttt{[i]} : t} \tag{9.4}$$

$$\frac{\tau(service) = \textbf{ServiceType}(\_,\_,\_) \quad \mathsf{ServiceInstType}(\top, \tau(service)) = t}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} \texttt{new}\ \ service\texttt{[}\star\texttt{]} : t} \tag{9.5}$$

Commandes La règle (9.6) vérifie la définition d'une commande par un service. Les types des arguments sont tout d'abord vérifiés comme appartenant bien aux types de données déclarés dans le modèle. La signature de la commande est ensuite recherchée dans le modèle. Le type de retour déclaré dans le modèle est comparé avec celui déclaré par le service. Le type déclaré par le service peut éventuellement être un sous-type du celui déclaré dans le modèle. L'appartenance de cette commande au type de service évalué est finalement vérifiée.

$$\frac{\begin{array}{c}\Gamma = (\Sigma, \Xi, \Delta) \\ \Gamma,\ \mu,\ \tau \vdash_{\mathrm{t}} type_i : t_i \qquad t_i = \mathsf{DataType}(\_), \forall i \in \{1, \ldots, n\} \\ \Xi^{-1}(cmdName, \langle t_1; \ldots; t_n\rangle) = \textbf{Command}(cmdLabel, methods) \\ methods(cmdName, \langle t_1; \ldots; t_n\rangle) = ret\_type \qquad \Gamma,\ \mu,\ \tau \vdash_{\mathrm{t}} type : t_d \\ \Gamma,\ \mu,\ \tau[id_1 \mapsto t_1, \ldots, id_n \mapsto t_n],\ ret\_type \vdash_{\mathrm{s}} cmpd : t_s \\ t_d = ret\_type \qquad t_s \sqsubseteq ret\_type \qquad \mathsf{ProvidedCommandType}(cmdLabel) \in \mu\end{array}}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{m}} \texttt{command}\ \ type\ \ cmdName\ \ \texttt{(}type_1\ \ id_1\texttt{,}\ \ \ldots\texttt{,}\ \ type_n\ \ id_n\texttt{)}\ \ cmpd} \tag{9.6}$$

La règle (9.7) vérifie la bonne utilisation d'une commande. Cette règle n'est valable que lorsque la commande est invoquée sur une unique instance de service. Si l'invocation de la commande a lieu sur un ensemble de services, la règle (9.8) est évaluée. L'analyse réalisée est la même, seul le type de retour diffère. La signature de la commande invoquée sert tout d'abord à rechercher la définition de la commande dans le modèle. Cette commande doit être requise par le type de service de l'implémentation en cours de vérification. Elle doit également être fournie par le type de service sur lequel elle est invoquée. Le type de retour de l'invocation correspond au type de retour déclaré dans le modèle.

$$\frac{\begin{array}{c}(\Sigma, \Xi, \Delta),\ \mu,\ \tau \vdash_{\mathrm{e}} exp : \mathsf{ServiceInstType}(i, \textbf{ServiceType}(\_, \mu', \_)) \\ i = 1 \\ (\Sigma, \Xi, \Delta),\ \mu,\ \tau \vdash_{\mathrm{e}} exp_j : t_j,\ \forall j \in \{1 \ldots n\} \\ \Xi^{-1}(cmdName, \langle t_1; \ldots; t_n\rangle) = \textbf{Command}(cmdLabel, methods) \\ t = methods(cmdName, \langle t_1; \ldots; t_n\rangle) \\ \mathsf{RequiredCommandType}(cmdLabel) \in \mu \qquad \mathsf{ProvidedCommandType}(cmdLabel) \in \mu'\end{array}}{(\Sigma, \Xi, \Delta),\ \mu,\ \tau \vdash_{\mathrm{e}} exp\texttt{.}\,cmdName\,\texttt{(}exp_1\texttt{,}\ \ \ldots\ \ \texttt{,}\ \ exp_n\texttt{)} : t} \tag{9.7}$$

$$\frac{\begin{array}{c}(\Sigma,\Xi,\Delta),\ \mu,\ \tau \vdash_{\text{e}} exp : \mathsf{ServiceInstType}(i, \mathbf{ServiceType}(\_,\mu',\_)) \\ i \in (\mathbb{N}^* \cup \{\top\}) \setminus \{1\} \\ (\Sigma,\Xi,\Delta),\ \mu,\ \tau \vdash_{\text{e}} exp_j : t_j,\ \forall j \in \{1 \ldots n\} \\ \Xi^{-1}(cmdName, \langle t_1; \ldots; t_n\rangle) = \mathbf{Command}(cmdLabel, methods) \\ t = methods(cmdName, \langle t_1; \ldots; t_n\rangle) \\ \mathsf{RequiredCommandType}(cmdLabel) \in \mu \qquad \mathsf{ProvidedCommandType}(cmdLabel) \in \mu'\end{array}}{(\Sigma,\Xi,\Delta),\ \mu,\ \tau \vdash_{\text{e}} exp\texttt{.}cmdName\texttt{(}exp_1\texttt{, ... ,}\ exp_n\texttt{)} : \mathsf{ListType}(t)} \tag{9.8}$$

Événements Les règles (9.9) à (9.11) décrivent les opérations permettant à un service de publier un événement. La règle (9.9) décrit la déclaration d'un événement qui permet d'étendre l'environnement τ du service. L'identifiant *id* est ainsi associé à la définition de l'événement : $\tau[id \mapsto \mathsf{ProvidedEventType}(userTypeLabel)]$. Pour ce faire, la définition *userTypePath* du type de données de l'événement est récupérée dans l'environnement Γ. Les propriétés déclarées par le développeur sont analysées vis à vis de celles déclarées dans le modèle. Finalement, la règle vérifie que le type de données de l'événement est un sous-type du type de données d'un événement devant être publié par ce service.

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{upa}} userTypePath : userType \\ userType = \mathbf{UserType}(userTypeLabel, \tau_{\mathsf{prop}}) \\ \Gamma,\ \mu,\ \tau, \tau_{\mathsf{prop}} \vdash_{\text{p}} properties \\ \exists userTypeLabel',\ \mathsf{ProvidedEventType}(userTypeLabel') \in \mu \\ \Gamma = (\_,\_,\Delta) \qquad \Delta(userTypeLabel) \sqsubseteq \Delta(userTypeLabel')\end{array}}{\begin{array}{l}\Gamma,\ \mu,\ \tau \vdash_{\text{d}} \texttt{event<}userTypePath\texttt{> \{}properties\texttt{\} }id\texttt{;} \\ : \tau[id \mapsto \mathsf{ProvidedEventType}(userTypeLabel)]\end{array}} \tag{9.9}$$

Une fois qu'un événement est déclaré, le service peut en créer une instance qu'il peut publier. La règle (9.10) crée une instance d'un événement à partir d'une déclaration de type ProvidedEventType. La règle (9.11) vérifie que l'instruction `publish` ne reçoit en argument que des événements qui peuvent être publiés, c'est-à-dire des expressions de type EventInstType et dont le type de service évalué est un fournisseur.

$$\frac{\tau(identifier) = \mathsf{ProvidedEventType}(etl)}{\Gamma,\ \mu,\ \tau \vdash_{\text{e}} \texttt{new }\ identifier : \mathsf{EventInstType}(etl)} \tag{9.10}$$

$$\frac{\begin{array}{c}\Gamma,\ \mu,\ \tau \vdash_{\text{e}} exp : \mathsf{EventInstType}(etl) \\ \mathsf{ProvidedEventType}(etl) \in \mu\end{array}}{\Gamma,\ \mu,\ \tau,\ t \vdash_{\text{s}} \texttt{publish (}exp\texttt{) ; } : \mathsf{Void}} \tag{9.11}$$

Les règles (9.12) à (9.14) vérifient les opérations relatives à la réception d'événements. Un service doit réaliser quatre étapes afin de recevoir des événements :

1. Déclarer le type de service qui publie l'événement qu'il désire recevoir,
2. Déclarer le type d'événement qu'il souhaite recevoir de ces services,
3. Définir un traitement à exécuter lorsqu'un événement est reçu,
4. Invoquer l'instruction `adopt` afin de sélectionner et activer un traitement particulier.

Le compilateur de services Pantaxou vérifie à chacune de ces étapes la cohérence des déclarations du développeur de service au sein d'un service ainsi que vis à vis du modèle d'environnement.

Nous avons vu précédemment la déclaration de service à la règle (9.3). L'étape suivante consiste à déclarer l'événement auquel souscrit le service. Cette déclaration précise deux éléments : d'une part le type de services qui émet l'événement, et d'autre part le nombre de souscriptions auprès des services de ce type. Le type de services est dénoté par le terme *service*, tandis que le nombre de souscriptions est dénoté par l'entier i. La règle (9.12) vérifie que le type de service auquel le service souscrit publie bien l'événement désiré. De plus, l'événement publié doit avoir un type de données associé qui soit un sous-type de l'événement désiré. Enfin, l'événement désiré par le développeur doit lui-même être un sous-type d'un événement requis par le service évalué tel que déclaré dans le modèle.

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{upa}} userTypePath : userType \\ userType = \mathbf{UserType}(userTypeLabel_1, \tau_{\mathsf{prop}}) \\ \Gamma,\ \mu,\ \tau, \tau_{\mathsf{prop}} \vdash_{\text{p}} properties \qquad \tau(service) = \mathbf{ServiceType}(\_, \mu', \_) \\ \exists userTypeLabel_2,\ \mathsf{ProvidedEventType}(userTypeLabel_2) \in \mu' \\ \exists userTypeLabel_3,\ \mathsf{RequiredEventType}(userTypeLabel_3) \in \mu \\ \Gamma = (\_, \_, \Delta) \\ \Delta(userTypeLabel_2) \sqsubseteq \Delta(userTypeLabel_1) \sqsubseteq \Delta(userTypeLabel_3)\end{array}}{\begin{array}{c}\Gamma,\ \mu,\ \tau \vdash_{\text{d}} \texttt{event<}userTypePath\texttt{> from } service\texttt{[}i\texttt{] \{}properties\texttt{\}}\ id\texttt{;} \\ : \tau[id \mapsto \mathsf{RequiredEventType}(userTypeLabel)]\end{array}} \tag{9.12}$$

La troisième étape consiste à définir le traitement qui sera exécuté à chaque réception d'un événement. La règle (9.13) récupère la déclaration d'événement précédente dans l'environnement et vérifie son type. Le type de l'identifiant doit être une déclaration de type RequiredEventType. Le type de données de l'événement sert alors à évaluer le bloc de code qui définit le traitement de l'événement.

$$\frac{\begin{array}{c}\tau(imType) = \mathsf{RequiredEventType}(t) \\ t' = \mathsf{EventInstType}(t) \\ \Gamma,\ \mu,\ \tau[im \mapsto t'],\ \mathsf{Void} \vdash_{\text{s}} cmpd : \mathsf{Void}\end{array}}{\begin{array}{c}\Gamma,\ \mu,\ \tau \vdash_{\text{d}} \texttt{onReceive}\ id\ \texttt{(}imType\ im\texttt{)}\ cmpd \\ : \tau[id \mapsto \mathsf{EventReceptionType}(imType)]\end{array}} \tag{9.13}$$

Finalement, le traitement responsable d'un événement donné est sélectionné avec l'instruction `adopt`. Cette instruction sert également à sélectionner les sessions entrantes. La règle (9.14) vérifie que l'expression en argument est bien soit de type EventReceptionType, soit de type SessionReceptionType.

$$\frac{\Gamma,\ \mu,\ \tau \vdash_{\text{e}} exp : \mathsf{EventReceptionType}(\_) + \mathsf{SessionReceptionType}(\_)}{\Gamma,\ \mu,\ \tau,\ t \vdash_{\text{s}} \texttt{adopt (}exp\texttt{) ;} : \mathsf{Void}} \tag{9.14}$$

Sessions Les sessions en Pantaxou ont une notation et une sémantique très similaire à celles des événements. Les règles relatives aux déclarations des sessions, la création d'instance de sessions, la découverte de services et la définition d'un traitement sont disponibles en annexe D

ainsi que l'ensemble de la sémantique de Pantaxou. Nous présentons ici les instructions et expressions spécifiques à la manipulation des sessions.

L'instruction `invite` permet d'émettre ou de rediriger une session. Elle prend en premier argument le service auquel est destiné la session. Le second argument est une instance de session. La règle (9.15) vérifie que le service en cours d'évaluation nécessite bien ce type de session et que le service distant fournit bien cette fonctionnalité. Finalement, chaque branche de l'instruction est vérifiée. La première branche correspond au cas où la session émise est acceptée par le service distant, tandis que la seconde branche correspond au cas où la session a été rejetée. Lorsque la session est acceptée, une session active est alors accessible aux développeurs.

$$\frac{\begin{array}{c}\Gamma,\ \mu,\ \tau \vdash_{\text{e}} exp_1 : \mathbf{ServiceInstType}(\_, modes, \_) \\ \Gamma,\ \mu,\ \tau \vdash_{\text{e}} exp_2 : \mathsf{SessionInstType}(userTypeLabel) \\ \mathsf{RequiredSessionType}(userTypeLabel) \in \mu \\ \exists userTypeLabel', \Delta(userTypeLabel') \sqsubseteq \Delta(userTypeLabel) \\ \wedge\ \mathsf{ProvidedSessionType}(userTypeLabel') \in modes \\ type = \mathsf{ActiveSessionType}(userTypeLabel) \\ \Gamma,\ \mu,\ \tau[id \mapsto type],\ t \vdash_{\text{s}} cmpd_{acc} : t_{acc} \qquad \Gamma,\ \mu,\ \tau,\ t \vdash_{\text{s}} cmpd_{rej} : t_{rej} \\ t_{acc} = t_{rej} = t' \end{array}}{\begin{array}{r}\Gamma,\ \mu,\ \tau,\ t \vdash_{\text{s}} \texttt{invite(}exp_1\texttt{,}\ \ exp_2\texttt{)\{accepted(}type\ id\texttt{)}\ \ cmpd_{acc} \\ \texttt{rejected()}\ \ cmpd_{rej}\ \ \texttt{\}} : t'\end{array}} \tag{9.15}$$

Une session active peut également être obtenue en acceptant une session entrante. Un bloc `onReceive` définit le traitement d'une session entrante, il définit également en argument une variable de type `SessionInstType`. L'expression `accept` permet d'accepter la session entrante et d'obtenir une référence sur la session active. La règle (9.16) modélise l'évolution de l'état de la session lors d'une acceptation.

$$\frac{\Gamma,\ \mu,\ \tau \vdash_{\text{e}} exp : \mathsf{SessionInstType}(userTypeLabel)}{\Gamma,\ \mu,\ \tau \vdash_{\text{e}} \texttt{accept (}exp\texttt{)} : \mathsf{ActiveSessionType}(userTypeLabel)} \tag{9.16}$$

Au contraire, l'instruction `reject`, décrite par la règle (9.17), permet de refuser une session entrante. Elle prend en argument une instance de session qui n'est pas encore active et produit le type Void.

$$\frac{\Gamma,\ \mu,\ \tau \vdash_{\text{e}} exp : \mathsf{SessionInstType}(\_)}{\Gamma,\ \mu,\ \tau,\ t \vdash_{\text{s}} \texttt{reject(}exp\texttt{);} : \mathsf{Void}} \tag{9.17}$$

9.2.3 Sémantique dynamique

La sémantique dynamique a été définie pour les services entités (Annexe D.5). Elle ne présente cependant pas d'originalité justifiant de la détailler. Sa rédaction a été réalisée en quelques semaines et a toutefois permis de prendre certaines décisions de conception très tôt dans le processus de développement de l'approche Pantaxou. Par exemple, la sémantique a mis en évidence le problème de la découverte de l'ensemble des services, c.-à-d. `[*]`, qui peut retourner une liste vide de services. Nous avons convenu qu'il s'agissait d'une erreur dynamique et un bloc de code `onError`, global au service, est exécuté.

9.3 Bilan

Nous avons présenté dans ce chapitre l'approche Pantaxou dédiée au développement de services entités. Cette approche se compose de deux langages : PantaEnv pour définir un modèle d'environnement et PantaLog pour définir des logiques de coordination de services. Le modèle d'environnement déclare un ensemble de services abstraits et les communications qui ont lieu entre les services. Ces communications se composent d'un mode de communications et de types de données. Il existe trois modes de communications en Pantaxou : commande, événement et session. Lorsqu'un modèle est cohérent, des analyses permettent de le valider. Des logiques de services entités sont ensuite développés pour un modèle valide. Chaque logique définit une instance d'un service abstrait d'un modèle. Une logique ne contient pas de référence explicite à d'autres services mais un mécanisme de découverte de services fondé sur des propriétés relatives aux services permet d'interagir avec les autres entités. Chaque logique est vérifiée vis à vis du modèle auquel elle appartient. L'analyseur contrôle les déclarations de services, l'utilisation des propriétés et l'ensemble des opérations relatives aux communications.

Chapitre 10

Mise en œuvre de Pantaxou

Dans ce chapitre nous décrivons notre implémentation du langage Pantaxou [MPCL08]. Nous précisons dans un premier temps le domaine de notre étude et rappelons le processus de développement. Nous présentons ensuite la chaîne de compilation et son utilisation lors du développement de services.

10.1 Domaine de l'étude

L'approche Pantaxou est fortement inspirée par les modes de communications possibles en SIP, cette approche est cependant indépendante du protocole de communications sous-jacent. La mise en œuvre de Pantaxou a conduit au développement d'un compilateur de modèles d'environnement Pantaxou nommé DiaGen [CJLP07, JLP+08, JPCK08] et d'un compilateur de logiques Pantaxou, pour les services entités. Le compilateur DiaGen repose sur plusieurs protocoles de communications. Il est notamment possible de générer, outre l'implémentation fondée sur le protocole SIP, une implémentation fondée sur RMI ou les services Web, au choix.

10.2 Processus de développement

Le processus de développement en Pantaxou commence par la définition d'un modèle d'environnement comprenant la hiérarchie des services du système ainsi que les commandes et les types de données relatifs aux communications entre les services. Un modèle PantaEnv est fournit au générateur DiaGen (à gauche, Figure 10.1) qui produit un intergiciel dédié au développement des services du domaine. Cet intergiciel en Java assure la gestion des services qui seront déployés et fournit des mécanismes pour la découverte de services. Plusieurs protocoles de communications peuvent alors être mis en œuvre. DiaGen permet la génération d'un intergiciel fondé au choix sur le protocole SIP, la technologie RMI ou les services Web. Le compilateur PantaGen (à droite de la figure) génère ensuite pour chaque service entité un programme Java qui s'intègre à cet intergiciel.

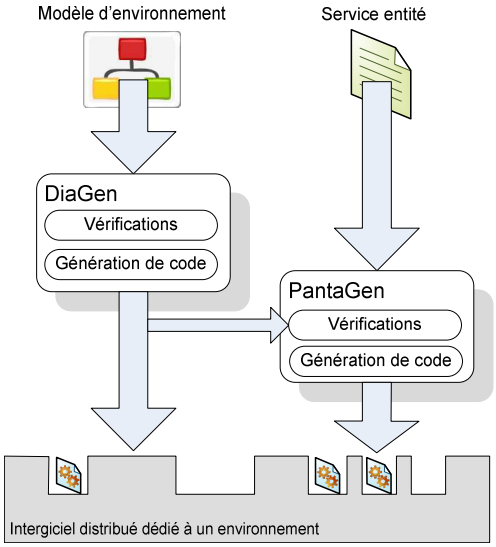


FIG. 10.1: Approche Pantaxou

10.3 Chaîne de compilation

La chaîne de compilation de l'approche Pantaxou se compose du compilateur DiaGen, d'un intergiciel généré par DiaGen pour un environnement donné et du compilateur de services PantaLog qui cible cet intergiciel.

10.3.1 DiaGen

Le format d'entrée de DiaGen, DiaSpec, est une extension de Java dérivée du langage de modélisation d'environnement PantaEnv. Un pré-traitement réécrit les modèles Pantaxou tels qu'ils ont été présentés au chapitre précédent en leur équivalent DiaSpec. Le compilateur DiaGen intègre les outils JastAdd [EH07b, HM03] et JastAddJ [EH07a]. Ces outils permettent de manipuler l'arbre abstrait de programmes Java. Après avoir analysé syntaxiquement un modèle d'environnement et vérifié sa cohérence, diverses classes Java sont générées pour chaque objet du modèle (c.-à-d. les services, les propriétés et les fonctionnalités des services, les commandes et les types de données). Les classes et les interfaces générées sont ensuite compilées avec un compilateur Java standard.

10.3.2 Intergiciel généré

L'intergiciel généré se compose de deux parties : une couche basse générique et une couche haute dédiée. La couche générique fournit les mécanismes de base pour les trois types de communications ainsi que des mécanismes génériques pour l'enregistrement des services dans le système et la découverte de services. La couche dédiée spécialise ces éléments à l'aide de classes Java abstraites et d'interfaces pour le modèle d'environnement considéré.

Il est possible à ce niveau de développer en Java des programmes pour l'intergiciel DiaGen généré. Un développeur de service DiaGen n'a qu'à implémenter la logique de son service dans une classe Java qui hérite de la classe abstraite correspondant au service qu'il souhaite implémenter. Le développeur dispose alors d'un enregistrement automatique de son service dans le

système, de mécanismes de découverte dédiés aux types de services avec lesquels il est amené à communiquer et de mécanismes de communications comme la publication d'événements. Ce niveau de programmation offre moins de support pour le développement de services qu'un développement en PantaLog. Il autorise cependant une plus large catégorie de services notamment grâce à la possibilité d'invoquer des bibliothèques Java arbitraires. Le développement des lecteurs Bluetooth présents dans le scénario de la secrétaire nécessite par exemple l'utilisation d'une bibliothèque de fonctions pour contrôler des ressources Bluetooth [Blu] et ont donc été développés en Java et intégrés dans l'intergiciel généré.

L'interface de programmation fournie par l'intergiciel ne varie pas. Il est par contre possible de choisir le protocole de communications mis en œuvre. Des implémentations RMI et SIP sont actuellement disponibles. Une troisième implémentation reposant sur les services Web est partiellement implémentée [1]. La technologie SOAP [kSO] issue des services Web a été mise en œuvre dans l'implémentation SIP et permet de construire la charge utile des messages SIP [JPCK08].

10.3.3 Compilateur de logiques Pantaxou

Le compilateur PantaGen prend en argument un service entité PantaLog et le modèle PantaEnv auquel il appartient. Après l'analyse syntaxique du programme, la cohérence du service vis à vis de son modèle est vérifié par PantaGen. L'utilisation des propriétés des services lors de la découverte de services est vérifiée. Les propriétés déclarées dans la logique doivent être définies dans le modèle. L'ensemble des opérations de communication entre les services sont également vérifiées. Si le service est valide, un programme Java est généré. Ce programme repose sur l'intergiciel dédié précédemment généré. La génération d'un service pour l'intergiciel dédié est possible car le compilateur PantaGen dispose non seulement du modèle d'environnement pour lequel un service PantaLog est écrit, mais également des motifs de génération de DiaGen. Les programmes Java ainsi produits s'intègrent dans l'intergiciel où ils sont exécutés.

La co-conception entre le langage PantaEnv pour la modélisation d'environnement et le langage PantaLog pour la logique permet de diminuer le pas de compilation, c'est-à-dire de réduire la complexité du compilateur PantaGen. Ce compilateur est ainsi plus aisé à maintenir car une partie de la complexité lors de la compilation d'un service est déportée dans le compilateur DiaGen. Les aspects de communications entre les services sont dévolus au compilateur de modèle, tandis que le compilateur de PantaLog réécrit les abstractions du langage en motifs de code Java fondés sur les objets et les classes définis par l'intergiciel généré.

10.4 Développement de services

Une implémentation partielle reposant sur les services Web a été développée dans le cadre du projet européen Amigo [Ami]. Cette implémentation s'intègre au reste de l'intergiciel Amigo qui fournit des services génériques pour la découverte, les commandes à distance et les événements. Le compilateur de modèle génère la surcouche dédiée à l'environnement ainsi que le squelette des services. Les programmes PantaLog sont ensuite compilés en un programme Java qui complète un squelette précédemment généré.

[1] Les sessions ne sont pas disponibles.

L'intégration de Pantaxou au projet Amigo a été testée avec l'exemple d'un gestionnaire de lumière contrôlant les lampes d'un bâtiment. Il est en charge de les allumer et de les éteindre en fonction de la luminosité. Ce scénario donne lieu à un modèle d'environnement (Figure 10.2) comprenant un type de données, la luminosité, une commande composée de deux méthodes et trois services. Le gestionnaire `LightManager` communique avec les différents périphériques, d'une part les capteurs de luminosité, d'autre part les lampes.

```
1  datatype Luminosity {
2    int luminosityVal;
3  }
4
5  command OnOff {
6    void on() ;
7    void off() ;
8  }
9
10 service Service {}
11
12 service Light extends Service {
13   provides command<OnOff>;
14 }
15
16 service LightSensor extends Service {
17   provides event<Luminosity>;
18 }
19
20 service LightManager extends Service {
21   required event<Luminosity>;
22   required command<OnOff>;
23 }
```

FIG. 10.2: Domaine pour la gestion de lampes en fonction de la luminosité

La génération de la couche dédiée pour l'intergiciel Amigo donne un ratio entre le code écrit et le code produit de l'ordre de 100 comme l'indique le tableau 10.1. L'implémentation Java de l'intergiciel Amigo repose sur le cadre de programmation OSGI [OSG]. Le compilateur de modèles génère un paquet OSGI *(bundle)* pour la couche dédiée ainsi qu'un squelette de paquet pour chaque service. Ces squelettes sont ensuite personnalisés manuellement ou bien une logique PantaLog y est incluse. L'utilisation du générateur sur un autre exemple indique que le ratio reste du même ordre de grandeur [Ami07].

	Description de domaine	Paquets Amigo générés		Ratio
		Couche dédiée	3 paquets squelettes	
Nb. de caractères	345	28 536	27 229	161
Nb. de mots	45	2 076	2 077	92
Nb. de lignes	23	974	958	84

TAB. 10.1: Ratio de génération pour le compilateur de modèles (implémentation pour la cible des services Web Amigo)

Lorsque l'intergiciel est généré, des développeurs spécifient des logiques particulières pour les services abstraits du modèle. Dans le cadre de ce scénario, un gestionnaire de lampes

a été développé en Pantalog (Figure 10.3). Il définit un cycle d'hystérésis autour des valeurs arbitraires 30 et 40 que produisent les capteurs de lumière. Si la luminosité devient trop faible, les lampes présentent dans le système sont découvertes puis allumées. Lorsque la luminosité augmente au-delà de 40, les lampes sont éteintes. Alors que le gestionnaire correspondant représente une trentaine de lignes de PantaLog, le compilateur de services génère près de quatre-vingt dix lignes de Java pour l'intergiciel dédié (Annexe E).

```
1  import command OnOff;
2  import datatype Luminosity;
3  import datatype Info;
4  import service LightManager;
5  import service Light;
6  import service LightSensor;
7
8  'light_manager@amigo.phoenix.labri.fr' instantiates LightManager {
9
10   service<Light> {
11   } light;
12
13   service<LightSensor> {
14   } lightSensor;
15
16   event<Luminosity> from lightSensor[*] {
17   } lumEvt;
18
19   onReceive lumEvtReception(lumEvt e) {
20     int lum = e.data.luminosityVal;
21     if (lum < 30) {
22       (new light[*]).on();
23     } else if (lum > 40) {
24       (new light[*]).off();
25     }
26   }
27    // ------- Main Section --------
28   initial {
29     adopt(lumEvtReception);
30   }
31 }
```

FIG. 10.3: Exemple de gestionnaire de lampes

10.5 Bilan

Nous avons décrit dans ce chapitre le processus de développement de services entités fondé sur l'approche Pantaxou. La concision des langages PantaEnv et PantaLog, pour spécifier un modèle et des programmes, favorise la compréhension des programmes et leur réutilisation. Cette concision ne se fait pas au détriment de l'expressivité. Les informations présentent dans un modèle d'environnement permettent la génération d'un intergiciel dédié. Il est actuellement possible de choisir parmi trois protocoles de communications pour cet intergiciel. Le langage de logiques Pantaxou s'appuie sur le modèle d'environnement et se concentre sur les aspects de coordination des communications entre les entités. Sa co-conception avec le langage PantaEnv permet de vérifier très tôt la validité d'un service vis à vis d'un modèle d'environnement donné. Grâce au haut niveau d'abstraction que procure l'intergiciel généré et à sa spécialisation pour

un environnement donné, le pas de compilation des services PantaLog est réduit vers celui-ci. Le compilateur PantaGen est ainsi plus simple à maintenir.

Troisième partie

Bilan

Chapitre 11

Apports

Les travaux que nous avons menés sur SPL puis Pantaxou présentent une approche globale pour le développement de services de communications. Ces deux langages ont été conçus à l'aide de l'approche des langages dédiés. La formalisation de leur sémantique a permis de les documenter avec un haut niveau d'abstraction quand les autres approches ne proposent qu'une documentation en langage naturel parfois accompagnée d'une implémentation de référence. SPL et Pantaxou proposent une approche langage au génie logiciel, et ce, dans le domaine des services de communications. La formalisation de ces langages a permis de définir des analyses qui garantissent la fiabilité des services ainsi développés.

11.1 Génie logiciel

La conception de DSL conduit à des langages simples d'utilisation car dédiés à un domaine. En simplifiant la programmation grâce à des abstractions de plus haut niveau que dans les langages généralistes, la communauté potentielle de développeurs s'agrandit. L'utilisation de SPL et Pantaxou est ainsi possible pour des programmeurs ayant une connaissance minimale de SIP et des communications. Dans le cadre de Pantaxou, l'expert qui modélise un environnement facilite la tâche des développeurs de services en explicitant son expertise.

11.1.1 Abstractions

La conception des langages SPL et Pantaxou nous a permis de définir quelques constructions originales. Les services de routage en SPL sont centrés sur la notion de sessions hiérarchiques avec un flot de contrôle explicite. Cette abstraction permet une gestion automatique de l'état entre deux événements à traiter. Le développement est ainsi simplifié pour les développeurs qui n'ont plus à expliciter ces opérations. Les services entités en Pantaxou sont conçus autour de trois modes de communications. Ces modes de communications sont des concepts de premier ordre pour la définition d'un environnement. Enfin un modèle PantaEnv capture l'hétérogénéité des services d'un environnement où l'approche Pantaxou peut être déployée, grâce notamment à l'utilisation d'une hiérarchie de services. Il est ainsi possible de définir un modèle de manière incrémentale. Le développement de services entités reste simple là où une approche généraliste nécessiterait un effort supplémentaire. Les développeurs devraient en effet mettre en œuvre leur logique particulière dans un environnement de développement générique. Ils leur faudraient développer manuellement l'équivalent de l'intergiciel dédié afin

de combler le manque d'abstraction inhérent à l'utilisation d'une approche généraliste. Dans l'approche Pantaxou, au contraire, cet intergiciel dédié est généré automatiquement grâce à DiaGen.

11.1.2 Évolutivité

Les communications constituent un domaine en perpétuelle évolution. De nouveaux périphériques communicants apparaissent régulièrement et les types de données qu'ils échangent évoluent également. SPL et Pantaxou répondent à ce besoin dans leur domaine respectif.

La fonctionnalité des modules externes de SPL permet de traiter l'ensemble des aspects n'ayant pas trait à la signalisation en dehors du service. Il est ainsi possible d'intégrer le système d'information d'une entreprise particulière sous la forme d'un module ou plusieurs modules. Le routage des appels peut alors être personnalisé en fonction d'informations issues de serveurs Web, d'agendas partagés ou de bases de données. L'appel d'un client important peut par exemple être routé directement vers le représentant commercial traitant son dossier, sans passer par le standard de l'entreprise.

Concernant Pantaxou, l'implémentation fondée sur SIP et générée par DiaGen permet le développement de services qui interagissent avec les périphériques SIP natifs, tels que les téléphones SIP et les passerelles téléphoniques (par exemple, Asterisk [Speb]). De plus, l'utilisation de technologies issues des services Web facilite les interactions entre ces derniers et les services SIP [JPCK08]. Enfin, l'intergiciel généré fournit du support de développement pour les services en périphérie du système qui communiquent avec le monde extérieur. Ce type de services ne peut pas être implémenté en PantaLog car ils nécessitent l'utilisation d'une API particulière, par exemple. C'est notamment le cas des détecteurs Bluetooth qui nécessite l'utilisation d'une pile de communications Bluetooth telle que BlueCove [Blu]. L'approche Pantaxou offre ainsi des perspectives d'évolution pour l'interopérabilité avec de nouveaux périphériques SIP, des services Web et d'autres technologies accessibles depuis Java. DiaGen facilite dans ce cas le développement d'adaptateurs entre les services existants de l'intergiciel DiaGen et les technologies extérieures.

11.1.3 Portabilité des services et réutilisation

Le haut niveau d'abstraction de SPL et Pantaxou permet d'être indépendant de la plateforme sous-jacente comme en attestent les différentes implémentations ou cibles de SPL et Pantaxou. Les abstractions que proposent SPL et Pantaxou permettent de gagner en concision. Les services sont plus simples à écrire et à comprendre. Les applications développées peuvent alors être portées avec des modifications mineures dans un nouvel environnement. La réutilisation de code est multiple. D'une part, les services développés profitent des évolutions ultérieures de leur environnement d'exécution. D'autre part, ils peuvent être rapidement adaptés pour être déployés dans un autre environnement, par exemple pour un nouvel utilisateur ou une nouvelle entreprise.

La disponibilité de la sémantique dynamique a quant à elle permis une implémentation directe d'un interprète comme l'illustre la règle traitant une requête `INVITE` initiale et son implémentation en O'Caml (Figure 11.1). L'implémentation O'Caml comporte sensiblement autant de fonctions qu'il y a de règles dans la sémantique, soit environ une centaine. L'implémentation du prototype O'Caml a ainsi été développée en une semaine.

$$\frac{\begin{array}{c} address = \langle service, rid, did \rangle \\ \mathsf{lookup\_branches}(\sigma, \mathsf{parent}(address)) = \langle branch \rangle \\ \mathsf{create\_session}(\phi, \sigma, address, \langle branch \rangle, \texttt{undialog}) = \sigma' \\ \mathsf{prepare\_method\_invocation}(\phi, \sigma', address, direction, \texttt{initial INVITE}) = \mu \\ \mu = \langle m, decls, stmt, envs, \sigma'' \rangle \\ \tau = \langle \sigma'', address \rangle \qquad r = \langle envs, \langle m, \langle rq, headers \rangle \rangle \rangle \end{array}}{\begin{array}{r} \langle \texttt{initial INVITE}(rid, did), direction, rq, headers \rangle, \phi, \sigma \models service \\ \Rightarrow \tau, r, \langle \texttt{INITIAL\_INVITE } \phi \rangle \models decls, stmt \end{array}}$$

11.1a. Définition de la règle sémantique

```
let interpret message phi sigma service =
  match message with
      (I_INVITE(rid, did), direction, rqid) ->
        let address = [service;(Reg, rid);(Dial, did)] in
        let branch = lookup_branches(sigma, parent(address)) in
        let sigma' = create_session(phi, sigma, address, branch, Some(If.UNINVITE)) in
        let (m_par, decls, stmts, envs, sigma'') =
            prepare_handler_invocation(phi, sigma', address, direction, If.INVITE) in
        let tau = (sigma'', address) in
        let rho = (envs, (m_par, rqid), []) in
          spl_handler_body tau rho ([]::[[T_INITIAL_INVITE(phi)]])
          (decls, stmts)
```

11.1b. Implémentation O'Caml

FIG. 11.1: Invocation de la méthode pour un premier INVITE

11.2 Fiabilité

Si nous considérons le langage SPL, la disponibilité de sa définition formelle et notamment de la sémantique statique fut d'une grande aide pour le développement de l'analyseur de programmes SPL. Elle a par exemple servi de fondation pour le développement de l'analyseur de types qui comporte environ 600 lignes de code O'Caml.

Dans le cadre des services de communications, SPL a permis d'améliorer la fiabilité des services de routage tout en préservant l'expressivité du langage. Il est notamment possible de vérifier que chaque requête produit bien une unique réponse vers l'appelant (à l'exception de la requête `ACK` qui ne doit rien renvoyer). L'analyse des expressions `forward` et `/SUCCESS` permet de garantir, respectivement, qu'un appel accepté n'est pas perdu, et qu'une réponse positive n'est pas fabriquée par un service de routage. Enfin, la manipulation correcte des en-têtes est garantie grâce aux constructions `when` et `with`.

La définition de la sémantique statique de Pantaxou a quant à elle permis les vérifications de cohérence entre les services d'un modèle d'une part et entre l'implémentation d'un service et son modèle de référence d'autre part.

L'approche des langages dédiés a permis d'améliorer le développement logiciel en supprimant des sources d'erreurs. La formalisation des langages dédiés rend notamment aisée la vérification de propriétés spécifiques au domaine. La fiabilité des services de communications est ainsi améliorée avec l'utilisation de SPL et Pantaxou vis à vis de services équivalents fondés sur des approches généralistes.

11.3 Bilan

L'utilisation de l'approche des langages dédiés nous a permis de concevoir deux langages, SPL et Pantaxou, dédiés au développement des services de communications. SPL est spécialisé pour le développement des services de routage tandis que Pantaxou généralise le développement des services de communications au développement des services entités. La conception de ces langages nous a permis de définir des abstractions adaptées et spécifiques aux services de communications modernes qui intègrent notamment la téléphonie et plus généralement les flux multimédia. Les solutions proposées offrent des mécanismes permettant leur évolution pour prendre en compte de futurs besoins. L'utilisation d'un langage dédié permet de porter les services développés sur différentes plates-formes, favorisant ainsi la réutilisation de code. Enfin, les abstractions fournies et la définition formelle des langages nous a permis de définir diverses analyses sur les services. Ces analyses améliorent à la fois la fiabilité des programmes et celle de la plate-forme sur laquelle ils s'exécutent.

Chapitre 12

Conclusion

La démocratisation d'Internet et la récente convergence des réseaux de télécommunications et des réseaux informatiques ont développé les communications entre les individus d'une part, et entre les équipements électroniques d'autre part. Les télécommunications sont désormais d'un usage courant comme l'eau ou l'électricité. La convergence des réseaux s'est accompagnée de l'apparition de protocoles informatiques permettant des communications audio et vidéo. Le développement de ces protocoles s'accompagne de développements de services. En effet, les utilisateurs des systèmes de téléphonie patrimoniaux ne sont disposés à migrer que s'ils retrouvent les services dont ils disposent déjà. Ces services doivent de plus avoir le même niveau de fiabilité. Parmi les protocoles de téléphonie IP, le protocole SIP est le plus flexible et le plus déployé par les opérateurs de téléphonie. Le protocole SIP permet plusieurs modes de communications. Outre les sessions définies pour les communications par flux de données, comme les communications audio et vidéo, le protocole SIP définit des communications par événements et envois de message. À l'instar du Web et du courrier électronique, la téléphonie SIP repose sur un protocole textuel propice aux développements de services. Les approches de développement de services existantes sont de deux types : d'une part celles fondées sur les langages généralistes, d'autre part celles fondées sur les langages dédiés. Cependant aucune approche existante n'est à la fois haut niveau, sûre et expressive. Soit il est nécessaire d'avoir un haut niveau d'expertise dans le protocole et la programmation informatique, soit les possibilités offertes sont particulièrement restreintes.

Nous avons présenté dans cette thèse une approche pour le développement de services de communications. Cette approche repose sur l'introduction de deux langages dédiés aux services de communications, nommés SPL et Pantaxou. Le langage SPL permet le développement de services de routage en spécifiant une logique pour le traitement des requêtes protocolaires SIP. Le langage Pantaxou généralise le développement des services de communications, et permet le développement de services entités. Contrairement aux services de routage qui ne font que réagir et modifier des stimuli de communications provenant d'entités communicantes, les services entités sont capables d'initier des stimuli vers d'autres services. Le langage Pantaxou exprime des logiques pour coordonner les communications des services entités. Ces logiques de coordinations sont elles-mêmes des services entités. Les langages SPL et Pantaxou offrent des abstractions dédiées aux services de communications. Leur compilateur respectif s'appuie sur ces abstractions pour analyser les services et déceler d'éventuelles incohérences. Les services valides sont ensuite déployés dans un environnement d'exécution.

Nous dressons dans ce chapitre un bilan des différentes contributions de cette thèse, puis

recensons un ensemble de perspectives de recherche.

12.1 Contributions

Les contributions de cette thèse se décomposent en plusieurs volets : l'analyse de domaine des services de communications, et des approches pour le développement de tels services. Les approches proposées concernent d'une part les services de routage, et d'autre part les services entités. Nous montrons dans chacun de ces deux cas la faisabilité et l'intérêt d'une approche langage. Nous présentons le processus complet de conception de langages dédiés aux services de communications, depuis l'analyse de domaine préalable jusqu'à l'implémentation.

Analyse de domaine Nous avons effectué une analyse des services de communications. Cette analyse a été complétée par l'analyse du domaine des services de communications fondés sur SIP et des méthodes de développement existantes. Nous avons ainsi pu définir les abstractions nécessaires pour un développement haut niveau, sûr et expressif des services de routage d'une part, et des services entités d'autre part. Enfin, nous avons également défini un ensemble de propriétés devant être garanties par les services.

Session Processing Language (SPL) À partir de l'analyse de domaine et du cahier des charges défini, nous avons conçu le langage SPL dédié au développement des services de routage. J'ai formellement défini ce langage, et dérivé de la sémantique des analyses ainsi qu'une implémentation. Les analyses garantissent des propriétés critiques du protocole SIP sous-jacent. Le compilateur vérifie par exemple qu'une réponse positive lors du processus de routage n'est pas créée ; qu'une réponse positive n'est pas perdue ; et, qu'une réponse est retournée pour toutes les requêtes.

Le langage fournit des abstractions pour les opérations de routage. Ces abstractions permettent une gestion automatisée de l'état d'un service entre les différents événements auxquels il réagit. Enfin, un système de modules permet l'interaction d'un service de routage avec le reste du système d'information. Le développement de services de routage avec le langage SPL est ainsi à la fois haut niveau, sûr et expressif par rapport aux approches existantes.

Pantaxou L'approche Pantaxou généralise le développement des services de communications aux services entités. Cette approche a été développée grâce à un processus de conception similaire à SPL. Le développement de services Pantaxou repose sur deux langages qui ont été coconçus. Un premier langage sert à spécifier un modèle de l'environnement où seront déployés les services. Ce modèle définit les services entités et les communications qu'ils mettent en œuvre. Un second langage permet de définir ensuite des logiques de coordinations des entités spécifiées dans l'environnement. Chaque logique est une instance d'un service entité défini dans le modèle.

Cette conception de services en deux temps permet de réaliser des vérifications très tôt dans le processus de développement. La cohérence d'un environnement est garantie avant l'implémentation d'une logique pour un service. Enfin, les logiques Pantaxou sont statiquement contrôlées vis à vis du modèle d'environnement dans lequel elles s'exécuteront. La définition d'une architecture distribuée est ainsi séparée du développement unitaire et local d'une logique.

Des compilateurs ont été développés pour ces deux langages. Le premier, DiaGen, génère un intergiciel Java dédié au modèle d'environnement fourni. Le second, PantaGen, génère, à

partir de la logique d'un service Pantaxou, un programme Java pour un intergiciel fourni par DiaGen.

12.2 Perspectives

Les travaux présentés proposent une nouvelle approche, fondée sur les langages dédiés, pour le développement de services de communications. Cette approche offre deux types de perspectives : d'une part dans le domaine des langages dédiés, et d'autre part pour le développement de services de communications. Les perspectives relatives aux services de communications concernent notamment des aspects non-fonctionnels et la détection d'interactions de services.

Langages dédiés Une perspective à court terme consiste à identifier des domaines connexes pour augmenter le langage. Nous pourrions par exemple intégrer des aspects langages dédiés pour l'interrogation de bases de données [VWKSB07] ou la manipulation de documents XML [BBK+07]. Cette intégration pourrait être intéressante aussi bien pour SPL que pour les logiques de coordination Pantaxou. Enfin, l'intégration de divers aspects dans le langage constituerait un cas d'étude pour les interprètes monadiques [ADM05, SBP99].

Une autre perspective consiste à appliquer l'approche des langages enchâssés [Hud96]. À l'instar des travaux d'intégration de BigWig [BMS02] dans Java, qui ont donné lieu à la conception de JWig [CM02], nous envisageons d'intégrer les concepts du langage de logiques Pantaxou dans le langage Java. Nous profiterons alors des différentes API Java disponibles. Le niveau de fiabilité introduit par le langage actuel de logiques devra cependant être préservé.

Aspects non-fonctionnels Une de nos prochaines directions de recherches pour Pantaxou consiste à intégrer dans le langage des aspects non fonctionnels des communications, comme la sécurité ou la qualité de service. Ces aspects seraient décrits au niveau du modèle d'environnement. L'analyse des logiques Pantaxou permettrait ensuite de certifier les implémentations vis à vis des nouvelles contraintes. Le langage E [Mil06], qui propose des mécanismes de sécurité pour les communications entre services, pourrait servir de base à ces travaux.

Interactions de services Une autre perspective intéressante s'inspire des travaux sur les interactions de services qui ont été réalisés pour les langages CPL [NLMK04] et LESS [WS05]. Une analyse des services SPL dans leur ensemble plutôt qu'individuelle permettrait de déterminer les conflits possibles à l'exécution. La détection des interactions pourrait être étudiée au niveau des services d'un même utilisateur, pour être étendue par la suite aux interactions entre les services de deux interlocuteurs situés dans des domaines Internet différents [KM07]. Des problèmes de confidentialité et de sécurité devront alors être pris en compte. Enfin, la vérification de modèle est une approche prometteuse pour la détection d'interactions [dB99] et pourrait être appliquée aux services SPL.

Les problèmes d'interactions de services sont également présents dans Pantaxou. Nous envisageons d'étudier comment détecter des conflits entre les logiques Pantaxou. Ces conflits peuvent survenir lorsque deux services envoient des ordres contradictoires à un troisième par exemple. L'analyse de cycles dans les communications pouvant conduire à un blocage ou un état erroné du système est également une perspective prometteuse.

12.3 Remarques finales

Dans le cadre de nos recherches sur le langage SPL, une perspective de recherche portant sur le développement visuel de services de routage nous a paru prometteuse. Les travaux de Latry [Lat07] ont visé à concevoir et développer l'atelier VisuCom. Cet éditeur de services permet de représenter graphiquement l'arbre de décision d'un service de routage manipulant un appel entrant. Les services produits sont ensuite compilés pour l'environnement d'exécution du langage SPL. Visucom et l'environnement d'exécution ont par la suite fait l'objet d'un dépôt logiciel et d'un dépôt de brevet [BCL+06c] en vue d'un transfert technologique dans la jeune pousse Siderion Technologies.

Le générateur d'intergiciels dédiés, conçu dans le cadre de nos recherches sur Pantaxou, fait actuellement l'objet d'un dépôt logiciel en vue d'une dissémination de la technologie auprès de partenaires industriels et académiques.

Publications

Publications liées au langage SPL

1. N. Palix, C. Consel, L. Réveillère, et J. Lawall. « A Stepwise Approach to Developing Languages for SIP Telephony Service Creation ». Dans *Principles, Systems and Applications of IP Telecommunications, IPTComm*, New-York, USA, pages 79–88, Juillet 2007.
2. L. Burgy, C. Consel, F. Latry, J. Lawall, N. Palix, et L. Réveillère. « Language Technology for Internet-Telephony Service Creation ». Dans *IEEE International Conference on Communications*, Istanbul, Turquie, pages 1795–1800, Juin 2006.
3. L. Burgy, C. Consel, F. Latry, N. Palix, et L. Réveillère. « A High-Level, Open Ended Architecture For SIP-based Services ». Dans *Proceedings of the 10th International Conference on Intelligence in service delivery Networks (ICIN 2006)*, Bordeaux, France, pages 364–365, Mai 2006. (Poster)
4. L. Burgy, C. Consel, F. Latry, L. Réveillère, et N. Palix. « Telephony over IP : Experience and Challenges ». *ERCIM News*, 63 :53–54, Octobre 2005.

Brevet et dépôt logiciel

1. Enregistrement d'un brevet européen et international : Dispositif d'interconnexion d'un système d'informations d'entreprise(s) à un serveur d'applications d'un système de téléphonie IP. Brevet INRIA, 06291276.1, enregistré le 7 août 2006, EP1887774.
2. Enregistrement logiciel auprès de l'Agence de Protection des Programmes (APP) du logiciel VisuCom et de son environnement exécution, version 1.0

Publications liées à l'approche Pantaxou

1. J. Mercadal, N. Palix, C. Consel et J. Lawall. « Pantaxou : a Domain-Specific Language for Developing Safe Coordination Services ». Dans *Seventh International Conference on Generative Programming and Component Engineering (GPCE)*, Nashville, Tennessee, USA, Octobre 2008. (À paraître)
2. W. Jouve, N. Palix, C. Consel et P. Kadionik. « A SIP-based Programming Framework for Advanced Telephony Applications ». Dans *The 2nd Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm'08)*, Heidelberg, Germany, Juillet 2008. Récompensé meilleur papier étudiant.

3. W. Jouve, J. Lancia, N. Palix, C. Consel, et J. Lawall. « High-level Programming Support for Robust Pervasive Computing Applications ». Dans *Proceedings of the 6th IEEE Conference on Pervasive Computing and Communications (PerCom'08)*, Hong Kong, China, pages 252–255, Mars 2008 (Poster).
4. C. Consel, W. Jouve, J. Lancia, et N. Palix. « Ontology-Directed Generation of Frameworks For Pervasive Service Development ». Dans *Proceedings of the 4th IEEE Workshop on Middleware Support for Pervasive Computing (PerWare'07)*, White Plains, New York, USA,pages 501–506, Mars 2007.

Bibliographie

[ABBC99] D.L. Atkins, T. Ball, G. Bruns, et K. Cox. « Mawl : A domain-specific language for form-based services ». *IEEE transactions on software engineering*, 25(3) :334–346, 1999.

[ADA+04] S. Andersen, A. Duric, H. Astrom, R. Hagen, W. Kleijn, et J. Linden. « Internet Low Bit Rate Codec (iLBC) », décembre 2004.

[ADM05] M.S. Ager, O. Danvy, et J. Midtgaard. « A Functional Correspondence between Monadic Evaluators and Abstract Machines for Languages with Computational Effects ». *Theoretical Computer Science*, 342(1) :149–172, 2005.

[Ami] « The IST European project Amigo ». http ://www.hitech-projects.com/euprojects/amigo/.

[Ami07] « Amigo D3.5 Amigo overall middleware : Final prototype implementation & documentation », 2007.

[Apa] Apache. « HTTP server project ». http ://www.apache.org.

[Ash02] P.J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann, 2002.

[Ass05] IEEE Standards Association. « IEEE 802.3 LAN/MAN CSMA/CD Access Method », 2005.

[ASU86] A.V. Aho, R. Sethi, et J.D. Ullman. *Compilers : principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1986.

[Aub07] R.J. Auburn. « Voice Browser Call Control : CCXML Version 1.0 (Working draft) ». Rapport Technique, W3C, janvier 2007.

[BBK+07] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, et A. Reilles. « *Tom manual* ». INRIA, nov 2007.

[BCL+06a] L. Burgy, C. Consel, F. Latry, J. Lawall, N. Palix, et L. Réveillère. « Language Technology for Internet-Telephony Service Creation ». Dans *IEEE International Conference on Communications*, pages 1795–1800, juin 2006.

[BCL+06b] L. Burgy, C. Consel, F. Latry, N. Palix, et L. Réveillère. « A High-Level, Open Ended Architecture For SIP-based Services ». Dans *Proceedings of the tenth International Conference on Intelligence in service delivery Networks (ICIN 2006)*, pages 364–365, Bordeaux, France, mai 2006.

[BCL+06c] L. Burgy, C. Consel, F. Latry, N. Palix, et L. Réveillère. « Routing device for an IP telephony system », août 2006. INRIA, 06291276.1.

[Ben86] J. Bentley. « Programming pearls : little languages ». *Commun. ACM*, 29(8) :711–721, 1986.

[Big] B. Biggs. « Kphone, a Free SIP User Agent ». http ://sourceforge.net/projects/kphone.

[BKVV08] M. Bravenboer, K.T. Kalleberg, R. Vermaas, et E. Visser. « Stratego/XT 0.17. A language and toolset for program transformation ». *Science of Computer Programming. Special Issue on Experimental Software and Toolkits (EST).(To appear).*, page 147, 2008.

[BLFM05] T. Berners-Lee, R. Fielding, et L. Masinter. « Uniform Resource Identifier (URI) : Generic Syntax ». Internet Engineering Task Force : RFC 3986, 2005.

[Blu] « BlueCove : Java library for Bluetooth ». http ://code.google.com/p/bluecove/. BlueCove group.

[BMS02] C. Brabrand, A. Møller, et M. I. Schwartzback. « The <bigwig> Project ». *ACM Transactions on Internet Technology*, 2(2) :79–114, mai 2002.

[BPM04] I.D. Baxter, C. Pidgeon, et M. Mehlich. « DMS® : Program Transformations for Practical Scalable Software Evolution ». *Proceedings of the 26th International Conference on Software Engineering-Volume 00*, pages 625–634, 2004.

[BPSM+06] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, et J. Cowan. « Extensible Markup Language (XML) 1.1 ». W3C Note REC-xml11-20060816, World Wide Web Consortium, 2006.

[BRLM07] L. Burgy, L. Réveillère, J. Lawall, et G. Muller. « A Language-Based Approach for Improving the Robustness of Network Application Protocol Implementations ». Dans *Proceeding of the 26th IEEE International Symposium on Reliable Distributed Systems*, pages 149–159, Beijing, China, octobre 2007. IEEE.

[BTKVV06] M. Bravenboer, M. Trygve Kalleberg, R. Vermaas, et E. Visser. « Stratego/XT : components for transformation systems ». Dans *PEPM '06 : Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 95–99. ACM, 2006.

[Bur08] L. Burgy. « *Approche langage au développement du support protocolaire d'applications réseaux* ». PhD thesis, Université de Bordeaux I - LaBRI / INRIA, avril 2008.

[CE00] K. Czarnecki et U.W. Eisenecker. *Generative programming : methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 2000.

[CJLP07] C. Consel, W. Jouve, J. Lancia, et N. Palix. « Ontology-Directed Generation of Frameworks For Pervasive Service Development ». Dans *Proceedings of The 4th IEEE Workshop on Middleware Support for Pervasive Computing (PerWare'07)*, pages 501–506, White Plains, NY, USA, mars 2007.

[CM98] C. Consel et R. Marlet. « Architecturing software using a methodology for language development ». Dans C. Palamidessi, H. Glaser, et K. Meinke, éditeurs, *Proceedings of the 10th International Symposium on Programming Language Implementation and Logic Programming*, volume 1490 de *Lecture Notes in Computer Science*, pages 170–194, Pisa, Italy, septembre 1998.

[CM02] A.S. Christensen et A. Møller. « JWIG User Manual », 2002.

[Con04] C. Consel. « *Domain-Specific Program Generation ; International Seminar, Dagstuhl Castle* », Chapitre From A Program Family To A Domain-Specific Language, pages 19–29. Numéro 3016 dans Lecture Notes in Computer Science, State-of-the-Art Survey. Springer-Verlag, 2004.

[Cri96] M. Crispin. « RFC 2060 : INTERNET MESSAGE ACCESS PROTOCOL — VERSION 4rev1 », décembre 1996.

[CRS+02] B. Campbell, J. Rosenberg, H. Schulzrinne, C. Huitema, et D. Gurle. « RFC3428 : Session Initiation Protocol (SIP) Extension for Instant Messaging », décembre 2002.

[dAFGD02] R. de Almeida Falbo, G. Guizzardi, et K.C. Duarte. « An ontological approach to domain engineering ». *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 351–358, 2002.

[dAMC+06] E.S. de Almeida, J.C.C.P. Mascena, A.P.C. Cavalcanti, A. Alvaro, V.C. Garcia, S.R. de Lemos Meira, et D. Lucrédio. « The Domain Analysis Concept Revisited : A Practical Approach ». *LECTURE NOTES IN COMPUTER SCIENCE*, 4039 :43, 2006.

[dB99] L. du Bousquet. « Feature Interaction Detection using Testing and Model-checking, Experience report ». Dans *World Congress on Formal Methods*, Toulouse, France, September 1999.

[DRM04] J. Deruelle, M. Ranganathan, et D. Montgomery. « Programmable Active Services for JAIN SIP ». Rapport Technique, National Institute of Standards and Technology, juin 2004.

[DTD] « Document Type Definition ». http ://www.w3schools.com/dtd/default.asp.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, et Anne-Marie Kermarrec. « The many faces of publish/subscribe ». *ACM Comput. Surv.*, 35(2) :114–131, 2003.

[EH07a] T. Ekman et G. Hedin. « The JastAdd extensible Java compiler ». Dans *OOPSLA '07 : Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 1–18, New York, NY, USA, 2007. ACM Press.

[EH07b] T. Ekman et G. Hedin. « The JastAdd system – modular extensible compiler construction ». *Science of Computer Programming*, 69(1-3) :14–26, 2007.

[Eug07] P. Eugster. « Type-based publish/subscribe : Concepts and experiences ». *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(1), 2007.

[FGM+97] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, et T. Berners-Lee. « RFC 2068 : Hypertext Transfer Protocol — HTTP/1.1 », janvier 1997. Status : PROPOSED STANDARD.

[FPDF98] W. Frakes, R. Prieto-Diaz, et C. Fox. « DARE : Domain analysis and reuse environment ». *Annals of Software Engineering*, 5 :125–141, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, et J. Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[GS08] P. Guenther et T. Showalter. « RFC5228 : Sieve : An Email Filtering Language », janvier 2008.

[HFW84] C. T. Haynes, D. P. Friedman, et M. Wand. « Continuations and Coroutines ». Dans *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 293–298, Austin, TX (USA), août 1984.

[HHKR89] J. Heering, PRH Hendriks, P. Klint, et J. Rekers. « The syntax definition formalism SDF–reference manual– ». *ACM SIGPLAN Notices*, 24(11) :43–75, 1989.

[HM03] G. Hedin et E. Magnusson. « JastAdd : an aspect-oriented compiler construction system ». *Science of Computer Programming*, 47(1) :37–58, 2003.

[Hoa69] C.A.R. Hoare. « An axiomatic basis for computer programming ». *Communications of the ACM*, 12(10) :576–580, 1969.

[Hud96] P. Hudak. « Building domain-specific embedded languages ». *ACM Computing Surveys*, 28 :196–196, 1996.

[Hud98] P. Hudak. « Modular domain specific languages and tools ». *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142, 1998.

[JLP+08] W. Jouve, J. Lancia, N. Palix, C. Consel, et J. Lawall. « High-level Programming Support for Robust Pervasive Computing Applications ». Dans *Proceedings of the 6th IEEE Conference on Pervasive Computing and Communications (PERCOM'08)*, pages 252–255, Hong Kong, China, mar 2008.

[Joh75] S. C. Johnson. « Yacc : Yet another compiler compiler ». Rapport Technique, Bell Telephone Laboratories, 1975.

[Jos03] S. Josefsson. « The Base16, Base32, and Base64 Data Encodings », juillet 2003.

[JPCK08] W. Jouve, N. Palix, C. Consel, et P. Kadionik. « A SIP-based Programming Framework for Advanced Telephony Applications ». Dans *The 2nd Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm'08)*, Heidelberg, Germany, jul 2008.

[JSR03] « JSR 116 : SIP Servlet API ». http ://jcp.org/en/jsr/detail ?id=116, mars 2003.

[JSR04] « JSR 22 : JAIN SLEE API Specification ». http ://jcp.org/en/jsr/detail ?id=22, mars 2004.

[JSR06] « JSR 32 : JAIN SIP API Specification ». http ://jcp.org/en/jsr/detail ?id=32, novembre 2006.

[Kal06] K. Kalleberg. « Stratego : a programming language for program manipulation ». *Crossroads*, 12(3) :4–4, 2006.

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, et A. S. Peterson. « Feature-Oriented Domain Analysis (FODA) Feasibility Study ». Rapport Technique, Carnegie-Mellon University Software Engineering Institute, November 1990.

[Ker82] B.W. Kernighan. « PIC-A Language for Typesetting Graphics ». *Software - Practice and Experience*, 12(1) :1–21, 1982.

[KM07] M. Kolberg et E.H. Magill. « Managing feature interactions between distributed SIP call control services ». *Computer Networks*, 51(2) :536–557, 2007.

[Knu64] D. Knuth. « Backus normal form vs. Backus Naur form ». *Commun. ACM*, 7(12) :735–736, 1964.

[kSO] « kSOAP 2 project ». http ://ksoap2.sourceforge.net.

[KT03] A. Kriegel et B Trukhnov. *SQL Bible*. Wiley, 2003.

[LAMS05] T. Leich, S. Apel, L. Marnitz, et G. Saake. « Tool support for feature-oriented software development : featureIDE : an Eclipse-based approach ». *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 55–59, 2005.

[Lan90] P.S. Langston. « Little Languages for Music ». *Computing Systems*, 3(2) :193–288, 1990.

[Lat07] F. Latry. « *Approche langage au développement logiciel : Application au domaine des services de Téléphonie sur IP* ». PhD thesis, Université de Bordeaux I - LaBRI / INRIA, septembre 2007.

[LCLL04] F. Liu, W. Chou, L. Li, et J. Li. « WSIP-Web service SIP endpoint for converged multimedia/multimodal communication over IP ». *Web Services, 2004. Proceedings. IEEE International Conference on*, pages 690–697, 2004.

[Ler97] X. Leroy. « The Objective Caml System Release 1.05 », 1997.

[LMB02] J.L. Lawall, G. Muller, et L.P. Barreto. « Capturing OS expertise in an event type system : the Bossa experience ». *Proceedings of the 10th workshop on ACM SIGOPS European workshop : beyond the PC*, pages 54–61, 2002.

[LS75] M. E. Lesk et E. Schmidt. « Lex - a lexical analyzer generator ». Rapport Technique, Bell Telephone Laboratories, 1975.

[LTM06] R. Lerdorf, K. Tatroe, et P. MacIntyre. *Programming PHP*. O'Reilly, 2006.

[LWS04] J. Lennox, X. Wu, et H. Schulzrinne. « RFC3880 : Call Processing Language (CPL) : A Language for User Control of Internet Telephony Services », octobre 2004.

[Mai] « MailMan, the GNU Mailing List Manager ». http ://www.list.org/. Free Software Foundation, Inc.

[Maj] « Majordomo ». http ://www.greatcircle.com/majordomo/. Great Circle.

[MGB+04] F. Mittlebach, M. Goossens, J. Braams, D. Carlisle, et C. Rowley. *The Latex Companion (Tools and Techniques for Computer Typesetting*. Addison-Wesley Professional, 2004.

[Mica] Microsoft. « Microsoft Office Communications Server 2007 ». http ://office.microsoft.com/en-us/communicationsserver/FX101729111033.aspx.

[Micb] Microsoft. « Microsoft Office Communications Server 2007 Server SDK ». http ://msdn.microsoft.com/en-us/library/bb632176.aspx.

[Micc] Microsoft. « Microsoft Response Point ». http ://www.microsoft.com/responsepoint/.

[Micd] Microsoft. « Microsoft SIP Processing Language ». http ://msdn.microsoft.com/en-us/library/bb632061.aspx.

[Mice] Sun Microsystem. « Enterprise JavaBeans Technology ». http ://-java.sun.com/products/ejb/.

[Micf] Sun Microsystem. « Java Servlet Technology ». http ://-java.sun.com/products/servlet/.

[Mil06] M.S. Miller. « *Robust Composition : Towards a Unified Approach to Access Control and Concurrency Control* ». PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, mai 2006.

[MMS00] D. Minoli, E. Minoli, et L. Sookchand. « ITU-T H.320/H.323 », 2000.

[Moi01] A. Moizard. « The GNU oSIP library ». http ://www.gnu.org/software/osip, juin 2001.

[Moi03] A. Moizard. « The eXtended Osip Library ». http ://savannah.nongnu.org/projects/exosip/, octobre 2003.

[MP99] V. Menon et K. Pingali. « A case for source-level transformations in MATLAB ». *Proceedings of the 2nd conference on Conference on Domain-Specific Languages*, 2 :5–5, 1999.

[MPCL08] J. Mercadal, N. Palix, C. Consel, et J. Lawall. « Pantaxou : a Domain-Specific Language for Developing Safe Coordination Services ». Dans *Seventh International Conference on Generative Programming and Component Engineering (GPCE)*, Nashville, Tennessee, USA, octobre 2008.

[MR96] J. Myers et M. Rose. « RFC 1939 : Post Office Protocol — Version 3 », mai 1996.

[MRC+00] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, et G. Muller. « Devil : An IDL for Hardware Programming ». Dans *4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 17–30, San Diego, California, octobre 2000.

[MSN06] « Unofficial MicroSoft Notification Protocol Version 15 documentation ». http ://msnpiki.msnfanatic.com/index.php/MSN_Protocol_Version_15, décembre 2006.

[Nei80] J. Neighbors. « *Software Construction Using Components* ». PhD thesis, University of California, Irvine, 1980.

[Nie04] A. Niemi. « Session Initiation Protocol (SIP) Extension for Event State Publication », octobre 2004.

[NLMK04] M. Nakamura, P. Leelaprute, K. Matsumoto, et T. Kikuno. « On detecting feature interactions in the programmable service environment of Internet telephony ». *Computer Networks (Amsterdam, Netherlands : 1999)*, 45(5) :605–624, August 2004.

[Nok] Nokia. « Sofia-SIP ». http ://opensource.nokia.com/projects/sofia-sip/.

[OAS07] « Open Document Format for Office Applications (OpenDocument) v1.1 », février 2007. OASIS Consortium.

[Opea] « OpenSER - the Open Source SIP Server ». http ://www.openser.org/.

[Opeb] « The OpenSIPStack Project ». http ://www.opensipstack.org/.

[OSG] « OSGI, "Open Service Gateway Initiative Homepage", http ://www.osgi.org. ».

[Par76] D.L. Parnas. « On the Design and Development of Program Families ». *IEEE Transactions on Software Engineering*, 2 :1–9, mars 1976.

[PCRL07] N. Palix, C. Consel, L. Réveillère, et J. Lawall. « A Stepwise Approach to Developing Languages for SIP Telephony Service Creation ». Dans *Proceedings of Principles, Systems and Applications of IP Telecommunications, IPTComm*, pages 79–88, New-York, NY, USA, juillet 2007. ACM Press.

[PD90] R. Prieto-Díaz. « Domain Analysis : An Introduction ». *Software Engineering Notes*, 15(2), avril 1990.

[Plo81] G. D. Plotkin. « A Structural Approach to Operational Semantics ». Rapport Technique DAIMI FN-19, University of Aarhus, 1981.

[POJK03] A. Pelinescu-Onciul, J. Janak, et Jiri Kuthan. « SIP Express Router (SER) ». *IEEE Network Magazine*, 17(4) :9, 2003.

[Pos81a] J. Postel. « RFC 791 : Internet Protocol », 1981.

[Pos81b] J. Postel. « RFC 793 : Transmission Control Protocol », 1981.

[Pos82] J. Postel. « RFC 821 : Simple Mail Transfer Protocol », 1982.

[Pri] B. Prijono. « pjsip.org – Open source SIP stack and media stack for presence, im/instant messaging, and multimedia communication ». http ://www.pjsip.org/.

[reS] « reSIProcate ». http ://www.resiprocate.org/. The reSIProcate project.

[Rév01] L. Réveillère. « *Approche langage au développement de pilotes de périphériques robustes* ». Thèse de doctorat, Université de Rennes 1, France, décembre 2001. Best Ph.D. Thesis award from ACM SIGOPS France.

[RLS99] J. Rosenberg, J. Lennox, et H. Schulzrinne. « Programming Internet Telephony Services ». *IEEE Internet Computing*, 3(3) :63–72, 1999.

[RM01] L. Réveillère et G. Muller. « Improving Driver Robustness : an Evaluation of the Devil Approach ». Dans *The International Conference on Dependable Systems and Networks*, pages 131–140, Göteborg, Sweden, juillet 2001. IEEE Computer Society.

[Roa02] A.B. Roach. « RFC3265 : Session Initiation Protocol (SIP)-Specific Event Notification », juin 2002.

[RSC+02] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, et E. Schooleral. « RFC3261 : SIP : Session Initiation Protocol », juin 2002.

[SA04] P. Saint-Andre. « RFC 3920 : Extensible Messaging and Presence Protocol (XMPP) : Core », octobre 2004.

[SBP99] T. Sheard, Z. Benaissa, et E. Pasalic. « DSL Implementation Using Staging and Monads ». Dans *Conference on Domain Specific Languages*, pages 81–94. USENIX, 1999.

[SCFJ03] H. Schulzrinne, S. Casner, R. Frederick, et V. Jacobson. « RFC3550 : RTP : A Transport Protocol for Real-Time Applications », juillet 2003.

[SCG+08] M. Spencer, B. Capouch, E. Guy, F. Miller, et K. Shumard. « IAX : Inter-Asterisk eXchange Version 2 – Work in progress », mars 2008.

[Sch86] D.A. Schmidt. *Denotational Semantics : a Methodology for Language Development.* Allyn and Bacon, Inc., 1986.

[SIM] « SIMPLE : SIP for Instant Messaging and Presence Leveraging Extensions ». http ://www.voip-telephony.org/rfc/simple. IETF SIMPLE Working group.

[SKG+07] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M.D. Corner, et E.D. Berger. « Eon : a language and runtime system for perpetual systems ». Dans *SenSys '07 : Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 161–174, New York, NY, USA, 2007. ACM.

[Sky] « Skype ». http ://www.skype.com.

[Spa] « The Apache SpamAssassin Project ». http ://spamassassin.apache.org/. Apache Fondation.

[Spea] « Speex : A Free Codec For Free Speech ». http ://speex.org/. The Xiph Open Source Community.

[Speb] A. Spencer. « Asterisk : The Open Source PBX ». http ://www.asterisk.org.

[Thi98] S. Thibault. « *Domain-Specific Languages : Conception, Implementation and Application* ». PhD thesis, University of Rennes 1, France, octobre 1998.

[Tra95] W. Tracz. « DSSA (Domain-Specific Software Architecture) : pedagogical example ». *ACM SIGSOFT Software Engineering Notes*, 20(3) :49–62, 1995.

[TTC95] R.N. Taylor, W. Tracz, et L. Coglianese. « Software development using domain-specific software architectures ». *ACM SIGSOFT Software Engineering Notes*, 20(5) :27–38, 1995.

[UML] « UML : Unified Modeling Language ». http ://www.uml.org/. Object Management Group (OMG).

[vD97] A. van Deursen. « "Domain-specific languages versus object-oriented frameworks : A financial engineering case study ». Dans *Proceedings Smalltalk and Java in Industry and Academia, STJA'97*, pages 35–39, Erfurt, Germany, septembre 1997. Ilmenau Technical University.

[vDK02] A. van Deursen et P. Klint. « Domain-specific language design requires feature descriptions ». *Journal of Computing and Information Technology*, 10(1) :1–17, 2002.

[VWKSB07] E. Van Wyk, L. Krishnan, A. Schwerdfeger, et D. Bodin. « Attribute grammar-based language extensions for Java ». *Proc. 21st ECCOP*, pages 575–599, 2007.

[W3C02] « XHTML 1.0 The Extensible HyperText Markup Language (Second Edition) ». http ://www.w3.org/TR/html/, août 2002. W3C HTML Working Group.

[Wei96] D.M. Weiss. « Family-oriented Abstraction Specification and Translation : the FAST Process ». Dans *Proceedings of the 11th Annual Conference on Computer Assurance (COMPASS), Gaithersburg, Maryland*, pages 14–22. IEEE Press, Piscataway, NJ, 1996.

[Wei98] D.M. Weiss. « Commonality Analysis :A Systematic Process for Defining Families ». *Lecture Notes in Computer Science*, 1429 :214, 1998.

[Wil04] D. Wile. « Lessons Learned from Real DSL Experiments ». *Sci. Comput. Program.*, 51(3) :265–290, 2004.

[WS00] X. Wu et H. Schulzrinne. « Where Should Services Reside in Internet Telephony Systems ? ». *IP Telecom Services Workshop (IPTS) 2000*, 2000.

[WS03] X. Wu et H. Schulzrinne. « Programmable End System Services Using SIP ». Dans *Proceedings of The IEEE International Conference on Communications 2003*. IEEE, 2003.

[WS05] X. Wu et H. Schulzrinne. « Handling Feature Interactions in the Language for End System Services. ». Dans *Feature Interactions in Telecommunications and Software Systems VIII, ICFI'05*, pages 270–287, Leicester, UK, juin 2005. IOS Press.

[XEP08] « XEP 166 : Jingle », février 2008. XMPP Standards Foundation.

[XSL] « XML Schema ». http ://www.w3.org/XML/Schema.

Quatrième partie

Annexes

Annexe A

SPL Syntax

A.1 Lexical Conventions

Blanks

The following characters are considered as blanks : space, newline and horizontal tabulation. Blanks separate adjacent identifiers, literals and keywords that would be otherwise confused as a single identifier, literal or keyword. Apart from that, they are ignored.

Comments

Comments are C-like comments. They start with the characters `/*` and end with the characters `*/`. C++ comments style can also be used ; all characters from the two characters `//` until the end of the line are considered as part of a comment. Comments are treated as blanks.

Special Lexeme

There are several kinds of identifiers :
- *ident* which defines identifiers (for variables, functions, branches and service name)
- *headerId* which defines SIP headers in the RFC fashion, which means, with a colon at the end of the header name
- *eventIdent* which defines SIP events in the RFC fashion, which means, an alphanumeric sequence of characters with possibly some '.' inside when template event packages are used
- *uriScheme* which is defined as *scheme* in the RFC 3261

They use common definitions of *letter* and *digit*.

letter ::= `A..Z | a..z`
digit ::= `0..9`

Identifiers

Identifiers are a sequence of letters, digits and _ (the underscore character) starting with a letter or an underscore. A letter can be any of the 52 lowercase and uppercase letters from the ASCII set. The current implementation places no limit on the number of characters of an identifier.

$$ident \;::=\; (letter \mid \_)\ (letter \mid digit \mid \_)^*$$

Header Names

Header names are a sequence of letters, digits and – (the minus character) starting with a letter and ending with a colon. A letter can be any of the 52 lowercase and uppercase letters from the ASCII set. The current implementation places no limit on the number of characters of a header name.

$$headerId \;::=\; \#letter\ (letter \mid digit \mid \_ \mid \texttt{!} \mid \texttt{\%} \mid \texttt{*} \mid \texttt{-} \mid \texttt{+} \mid \texttt{'} \mid \texttt{`} \mid \texttt{\~{}})^*\ \texttt{:}$$

Event Identifiers

Event identifiers are a sequence of letters, digits and special characters as defined in RFC 3265. A letter can be any of the 52 lowercase and uppercase letters from the ASCII set. The current implementation places no limit on the number of characters of an event identifier.

$$\begin{array}{lll} eventIdent & ::= & event\text{-}package\ (\texttt{.}\ event\text{-}template)^* \\ event\text{-}package & ::= & token\text{-}nodot \\ event\text{-}template & ::= & token\text{-}nodot \\ token\text{-}nodot & ::= & (letter \mid digit \mid \_ \mid \texttt{!} \mid \texttt{\%} \mid \texttt{*} \mid \texttt{-} \mid \texttt{+} \mid \texttt{'} \mid \texttt{`} \mid \texttt{\~{}})^+ \end{array}$$

URI Scheme

URI scheme are defined as in RFC 3261

$$uriScheme \;::=\; letter\ (letter \mid digit \mid \texttt{-} \mid \texttt{+} \mid \texttt{.})^*$$

Integer Literals

An integer literal is a sequence of one or more digits. Integer literals are in decimal (radix 10).

$$integer \;::=\; \texttt{0} \mid (\texttt{1..9}\ digit^*)$$

Prefix and Infix Operator

The following tokens are the SPL operators :

```
=    +    -       *          /    &&
==   !=   <       >          >=   ||
<=   !    match   notmatch
```

Keywords

The identifiers below are reserved keywords :

```
BODY        bool        branch      break       bye
cancel      case        continue    default     deploy
dialog      else        event       extern      false
FIFO        foreach     forward     http        https
if          in          incoming    int         LIFO
mailto      match       notmatch    outgoing    parallel
pop         processing  push        reason      registration
request     requestURI  response    return      select
service     sip         sips        string      this
time        true        type        undeploy    uninvite
unregister  unsubscribe uri         void        when
with
```

The following character sequences are also reserved.

```
{  }   [  ]   (  )  <  >  |
=   :  ,  ;  .
```

Besides the above mentioned keywords, all *sipHeader*, *responseKind* and *SIPmethod* are also keywords and are written in uppercase.

Ambiguities

Lexical ambiguities are resolved according to the "longest match" rule : when a character sequence can be decomposed into tokens in several different ways, the resulting decomposition is the one with the longest first token.

A.2 SPL Program

A SPL program consists of a service name and a *service* construct.

program ::= `service` *ident* { *service* }
service ::= `processing` { *declarations*$^?$ *session** }
| *declarations*$^?$ *session**
session ::= `registration` { *declarations*$^?$ *session** }
| `dialog` { *declarations*$^?$ *method*$^+$ }
| `event` *eventIdent* { *declarations*$^?$ *method*$^+$ }
| *method*
method ::= *typExp* *direction*$^?$ *methodName* (*arg*$^?$) { *stmt*$^+$ }
| *typExp* *direction*$^?$ *methodName* (*arg*$^?$) { *branch*$^+$ }
branch ::= `branch` *ident* (| *ident*)* { *stmt*$^+$ }
| `branch default` { *stmt*$^+$ }
direction ::= `incoming`
| `outgoing`
methodName ::= *SIPmethod*
| *ctrlMethod*
SIPmethod ::= `ACK`
| `BYE`
| `CANCEL`
| `INVITE`
| `MESSAGE`
| `NOTIFY`
| `OPTIONS`
| `PUBLISH`
| `REGISTER`
| `REINVITE`
| `REREGISTER`
| `RESUBSCRIBE`
| `SUBSCRIBE`
ctrlMethod ::= `deploy`
| `undeploy`
| `uninvite`
| `unregister`
| `unsubscribe`

Notes :
- Without explicit *direction*, a method applies for both direction, incoming and outgoing.
- There is no SIP message associated with control methods, so headers are not accessible. These methods do not have a response return type but a `void` one.
- `unregister` is called after a `REGISTER` with a zero `Expire` field or after related timeout.
- `unsubscribe` is called after a `SUBSCRIBE` with a zero `Expire` field, a `NOTIFY` with `Subscription-State: terminated` header or after related timeout.
- `deploy` and `undeploy` can only appear directly in the processing block.
- Processing block can have only one registration block.

- Registration block can have only one session block.
- Registration block can have multiple event block, but each one must have an unique event identifier.

A.3 Type Expressions

```
typExp      ::= simpleType
              | modifier? simpleType<integer?>
              | ident
simpleType  ::= void
              | bool
              | int
              | request
              | response
              | string
              | time
              | uri
modifier    ::= LIFO
              | FIFO
```

Notes :

- The sequence size must be fixed at compile time. A policy could be used to limit the size value.
- According to static properties, some information about a sequence (*e.g.* maximum size, starting position, current position) could be forgotten.
- The sequence head is on the left side, this is the first element to pop. According to the sequence kind, the push operation add an element on either the head or the tail of the sequence.
- If the modifier is not used, a sequence is a `LIFO` sequence.

A.4 Function Call

```
functionCall ::= ident ( expList? )
expList      ::= exp ( , exp)*
```

A.5 Known URI Kinds

```
uriKind ::= https
          | http
          | mailto
          | sips
          | sip
          | uri
          | uriScheme
```

A.6 Declarations

```
declarations ::= declarations declaration
               | declaration
declaration  ::= typExp ident (= exp)? ;
               | extern typExp ident(args?) ;
               | typExp ident(args?) {stmt+}
               | type ident { (typExp ident ;)+} ;
args         ::= arg (, arg)*
arg          ::= typExp ident
```

Notes :
- Initialize an identifier with a `forward` expression is only allowed in a method, this is a signaling action. So, it cannot happen in blocks like `processing`, `registration` and `session` outside of a method.
- A function can only call another one which is already declared.

A.7 Statements

```
compound     ::= stmt
               | {stmt+}
stmt         ::= place = exp ;
               | declaration
               | return exp? (branch ident)? ;
               | if(exp) compound
               | if(exp) compound else compound
               | when exp (when-headers) compound
               | when exp (when-headers) compound else compound
               | with ;
               | foreach(ident in exp) compound
               | select(exp) { selectCase* selectDefault? }
               | functionCall ;
               | continue ;
               | break ;
               | push place exp ;
when-headers ::= when-header (, when-header)*
when-header  ::= headerId typExp ident (<- const)?
selectCase   ::= case orList : (stmt)*
selectDefault ::= default : (stmt)*
orList       ::= const (| const)*
place        ::= ident(.ident)*
               | sipHeader
```

Notes :
- `continue` and `break` are only allowed inside a `foreach`.
- *functionCall* is the only expression that can also be a statement.
- The non-terminal *place*, is a place you can assign into, like a variable or a header field in a request.

– Header fields, in either a response or a request, are modified by the `with` construct.
– The expression in `when` can only be an identifier.

A.8 Message Modification

The `with` construct could be used as an statement with the `this` expression which denote the current processed request, or as an expression with a response value.

```
with         ::= exp with {messageField (, messageField)*}
messageField ::= reason=exp
               | headerId exp
```

A.9 Expressions

```
exp    ::= ident(.ident)*
         | constant
         | sipHeader
         | unop exp
         | exp binop exp
         | (parallel)? forward (exp)?
         | with
         | (exp)
         | reason
         | BODY
         | requestURI
         | this
         | pop place
         | functionCall
unop   ::= !
         | -
binop  ::= +
         | -
         | *
         | /
         | <
         | >
         | ==
         | !=
         | <=
         | >=
         | &&
         | ||
         | match
         | notmatch
```

```
constant  ::= true
            | false
            | integer
            | "string"
            | uri
            | sequence
            | response
sequence  ::= <constant (, constant)*>
            | <>
uri       ::= 'uriKind :uriString'
```

Notes :
- The expression, used in `with` expression, must be of response or request kind.
- The expression, used in `pop` expression, must be of sequence kind.
- `match` and `notmatch` use POSIX regular expressions.

A.10 SIP Headers

If SIP headers, specified in the RFC 3261 and RFC 3265, are mandatory in at least one method, they are integrated in the SPL compiler as keywords. SIP headers are restricted in terms of type or access rights (NA, RO, RW). Header keywords refer to request headers only. When a header is not mandatory in a request and for all responses, the `when` statement allow a protected access to read headers in messages. Headers can only be modified with the `with` expression.

```
sipHeader ::= CALL_ID
            | CONTACT
            | CSEQ
            | EVENT
            | FROM
            | MAX_FORWARDS
            | SUBSCRIPTION_STATE
            | TO
            | VIA
```

A.11 Constant Responses

Standard responses in RFC 3261 and RFC 3265 are defined in the language.

```
response         ::= /ERROR responseError?
                   | /SUCCESS responseSuccess?
responseSuccess  ::= /OK
                   | /ACCEPTED
responseError    ::= /CLIENT clientErr?
                   | /GLOBAL globalErr?
                   | /REDIRECTION redirectionErr?
                   | /SERVER serverErr?
redirectionErr   ::= /ALTERNATIVE_SERVICE
                   | /MOVED_PERMANENTLY
                   | /MOVED_TEMPORARILY
                   | /MULTIPLE_CHOICES
                   | /USE_PROXY
```

```
clientErr  ::= /ADDRESS_INCOMPLETE
             | /AMBIGUOUS
             | /BAD_EXTENSION
             | /BAD_REQUEST
             | /BUSY_HERE
             | /CALL_OR_TRANSACTION_DOES_NOT_EXIST
             | /EXTENSION_REQUIRED
             | /FORBIDDEN
             | /GONE
             | /INTERVAL_TOO_BRIEF
             | /LOOP_DETECTED
             | /METHOD_NOT_ALLOWED
             | /NOT_ACCEPTABLE_HERE
             | /NOT_ACCEPTABLE
             | /NOT_FOUND
             | /PAYMENT_REQUIRED
             | /PROXY_AUTHENTICATION_REQUIRED
             | /REQUESTURI_TOO_LONG
             | /REQUEST_ENTITY_TOO_LARGE
             | /REQUEST_PENDING
             | /REQUEST_TERMINATED
             | /REQUEST_TIMEOUT
             | /TEMPORARILY_UNAVAILABLE
             | /TOO_MANY_HOPS
             | /UNAUTHORIZED
             | /UNDECIPHERABLE
             | /UNSUPPORTED_MEDIA_TYPE
             | /UNSUPPORTED_URI_SCHEME
serverErr  ::= /BAD_GATEWAY
             | /MESSAGE_TOO_LARGE
             | /NOT_IMPLEMENTED
             | /SERVER_INTERNAL_ERROR
             | /SERVER_TIMEOUT
             | /SERVICE_UNAVAILABLE
             | /VERSION_NOT_SUPPORTED
globalErr  ::= /BUSY_EVERYWHERE
             | /DECLINE
             | /DOES_NOT_EXIST_ANYWHERE
             | /NOT_ACCEPTABLE
```

Annexe B

SPL Semantics

This document describes the semantics of aspects of the SPL language relating to SIP RFC 3261.

B.1 Simplified Syntax

The semantics considers a much simplified version of the SPL language. An SPL program has the following form.

```
service service {
  processing {
    (type x = exp ;)*
    (direction? type method_name () { (branch branch {decl* stmt})+ })*
    registration {
      (type x = exp ;)*
      (direction? type method_name () { (branch branch {decl* stmt})+ })*
      dialog {
        (type x = exp ;)*
        (direction? type method_name () { (branch branch {decl* stmt})+ })*
}}}}
```

The grammar is defined as follows :

program	::=	`service` *service* {`processing` {*decl** *method** `registration` {*decl** *method** `dialog` {*decl** *method** }}}}
decl	::=	*type* *x* = *exp* `;`
method	::=	*direction*? *type* *method_name* { (`branch` *branch* {*decl** *stmt*})+ }
direction	::=	`incoming` | `outgoing`
type	::=	`response` | `uri` | `uri list` | `void` | `bool` | `string` | `status`
stmt	::=	{} | {$stmt_1$ $stmt_2$} | *x* = *exp* | `return` *exp*? `branch` *b* `;` | `if (`*exp*`)` $stmt_1$ `else` $stmt_2$ | exp_1 `with` *field* = exp_2;
exp	::=	`forward` *exp* | `parallel forward` *exp* | < *exp* > | exp_1 == exp_2 | *x* | *c* | `requestURI` | exp_1 `with` *field* = exp_2 | *this*

service, *branch*, and x are arbitrary identifiers. The $\{\, decl^{+}\ \ stmt\}$ form is only allowed at the root of a method handler. *method_name* is restricted according to the level at which it occurs :

- At the `service` level, the allowed methods are `deploy` and `undeploy`.
- At the `registration` level, the allowed methods are `REGISTER`, `REREGISTER`, and `unregister`.
- At the `dialog` level, the allowed methods are `INVITE`, `REINVITE`, `CANCEL`, `ACK`, `BYE`, and `end_dialog`.

Constants c are strings, responses (*e.g.*, `/SUCCESS`, `/ERROR`, and various refinements thereof), and lists of uris (*e.g.*, `'sip :joe@bigcorp.com'`). `status` is an enumerated type with elements `SUCCESS` and `ERROR`.

B.2 Static Semantics

The only types *ty* used by the type system are request, response, uri, uri list, void, bool, string, and status. The type system is defined in terms of an environment τ of type $var \rightarrow ty$. The judgments for statements, expressions, declarations, method declarations and programs have the following form, where $t_0, t \in ty$:

- $t_0, \tau \vdash stmt : t$
- $t_0, \tau \vdash exp : t$
- $t_0, \tau \vdash decl : \tau'$
- $\tau \vdash method$
- $\vdash program$

t_0 is always the type of the enclosing method, and is used to determine the return type of `forward`. For example, there is no response associated with an `ACK`, and so a `forward` in an `ACK` handler has return type void.

The judgments for statements are defined as follows :

$$t_0, \tau \vdash \{\,\} : \mathsf{void} \qquad \frac{t_0, \tau \vdash stmt_1 : \mathsf{void} \qquad t_0, \tau \vdash stmt_2 : t}{t_0, \tau \vdash \{\, stmt_1 \ \ stmt_2\,\} : t}$$

$$\frac{t_0, \tau \vdash exp : t \qquad t = \tau(x)}{t_0, \tau \vdash x \ = \ exp : \mathsf{void}}$$

$$\frac{t_0, \tau \vdash exp : t}{t_0, \tau \vdash \texttt{return}\ \ exp\ \ \texttt{branch}\ \ b\ \texttt{;} : t} \qquad t_0, \tau \vdash \texttt{return branch}\ \ b\ \texttt{;} : \mathsf{void}$$

$$\frac{t_0, \tau \vdash exp : \mathsf{bool} \qquad t_0, \tau \vdash stmt_1 : t_1 \qquad t_0, \tau \vdash stmt_2 : t_2 \qquad t_1 = t_2}{t_0, \tau \vdash \texttt{if}\ \ (exp)\ \ stmt_1\ \ \texttt{else}\ \ stmt_2 : t_1}$$

$$\frac{t_0, \tau \vdash exp_1 : \mathsf{request} \qquad t_0, \tau \vdash exp_2 : t_1 \qquad t_0, \tau \vdash exp_2 : t_2 \qquad t_1 = t_2}{t_0, \tau \vdash exp_1\ \ \texttt{with}\ \ field\ =\ \ exp_2\ \texttt{;} : \mathsf{void}}$$

The judgments for expressions are defined as follows :

$$\frac{t_0, \tau \vdash exp : \mathsf{uri\ list}}{t_0, \tau \vdash \texttt{forward}\ \ exp : t_0} \qquad \frac{t_0, \tau \vdash exp : \mathsf{uri\ list}}{t_0, \tau \vdash \texttt{parallel forward}\ \ exp : t_0}$$

$$\frac{t_0, \tau \vdash exp : \mathsf{uri}}{t_0, \tau \vdash <\ \ exp\ \ > : \mathsf{uri\ list}} \qquad t_0, \tau \vdash \texttt{requestURI} : \mathsf{uri}$$

$$\frac{c \in \{\texttt{"..."}, \ldots\}}{t_0, \tau \vdash c : \mathsf{string}} \qquad \frac{c \in \{\texttt{<'sip :joe@bigcorp.com',...>}, \ldots\}}{t_0, \tau \vdash c : \mathsf{uri\ list}}$$

$$\frac{t_0, \tau \vdash exp_1 : t_1 \qquad t_0, \tau \vdash exp_1 : t_2 \qquad t_1 = t_2}{t_0, \tau \vdash exp_1\ \ == \ \ exp_2 : \mathsf{bool}} \qquad t_0, \tau \vdash x : \tau(x)$$

$$\frac{c \in \{\texttt{SUCCESS}, \texttt{ERROR}, \ldots\}}{t_0, \tau \vdash c : \mathsf{status}} \qquad \frac{c \in \{\texttt{/SUCCESS}, \texttt{/ERROR}, \ldots\}}{t_0, \tau \vdash c : \mathsf{response}}$$

$$\frac{t_0, \tau \vdash exp_1 : \mathsf{response} \qquad t_0, \tau \vdash exp_2 : \mathsf{string}}{t_0, \tau \vdash exp_1\ \ \texttt{with}\ \ \mathit{field}\ \ \texttt{=}\ \ exp_2 : \mathsf{response}} \qquad t_0, \tau \vdash \texttt{this} : \mathsf{request}$$

The judgments for declarations are defined as follows, in which we use *type* as either the syntactic type name or the name of the corresponding type in the type system, as convenient. The handler return type argument to the analysis of each expression is $\perp$ because declarations are not allowed to use `(parallel) forward`.

$$\frac{\perp, \tau \vdash exp : t \qquad t = type}{\tau \vdash type\ \ x\ \ \texttt{=}\ \ exp\ \texttt{;} : \tau[x \mapsto type]}$$

$$\frac{\perp, \tau \vdash exp : t \qquad t = type \qquad \tau[x \mapsto type] \vdash decl_2 \ldots : \tau'}{\tau \vdash type\ \ x\ \ \texttt{=}\ \ exp\ \texttt{;}\ \ decl_2\ \ \texttt{...} : \tau'}$$

The judgments for method declarations are defined as follows :

$$\frac{\begin{array}{c}\tau \vdash decl_{1_1} \ldots decl_{1_{m_1}} : \tau_1 \qquad \mathsf{status}, \tau_1 \vdash stmt_1 : t_1 \\ \ldots \\ \tau \vdash decl_{n_1} \ldots decl_{n_{m_n}} : \tau_n \qquad \mathsf{status}, \tau_n \vdash stmt_n : t_n \\ t_1 = \ldots = t_n = \mathsf{void}\end{array}}{\begin{array}{l}\tau \vdash direction^?\ \ \texttt{void}\ \ \mathsf{deploy} \\ \quad \{\ \ \texttt{branch}\ \ branch_1\ \ \{ decl_{1_1} \ldots decl_{1_{m_1}}\ \ stmt_1\ \ \} \\ \quad\ \ \ldots \\ \quad\ \ \texttt{branch}\ \ branch_n\ \ \{ decl_{n_1} \ldots decl_{n_{m_n}}\ \ stmt_n\ \ \}\}\end{array}}$$

$$\frac{\begin{array}{c} method\_name \in \{\texttt{deploy}, \texttt{undeploy}, \texttt{unregister}, \texttt{end\_dialog}\} \\ \tau \vdash decl_{1_1} \ldots decl_{1_{m_1}} : \tau_1 \qquad \mathsf{void}, \tau_1 \vdash stmt_1 : t_1 \\ \ldots \\ \tau \vdash decl_{n_1} \ldots decl_{n_{m_n}} : \tau_n \qquad \mathsf{void}, \tau_n \vdash stmt_n : t_n \\ t_1 = \ldots = t_n = \mathsf{void} \end{array}}{\begin{array}{l} \tau \vdash direction^? \ \texttt{void}\ method\_name \\ \quad \{\ \texttt{branch}\ branch_1\ \{ decl_{1_1} \ldots decl_{1_{m_1}}\ stmt_1\ \} \\ \quad\ \ldots \\ \quad\ \texttt{branch}\ branch_n\ \{ decl_{n_1} \ldots decl_{n_{m_n}}\ stmt_n\ \}\} \end{array}}$$

$$\frac{\begin{array}{c} \tau \vdash decl_{1_1} \ldots decl_{1_{m_1}} : \tau_1 \qquad \mathsf{void}, \tau_1 \vdash stmt_1 : t_1 \\ \ldots \\ \tau \vdash decl_{n_1} \ldots decl_{n_{m_n}} : \tau_n \qquad \mathsf{void}, \tau_n \vdash stmt_n : t_n \\ t_1 = \ldots = t_n = \mathsf{void} \end{array}}{\begin{array}{l} \tau \vdash direction^? \ \texttt{void ACK ()} \\ \quad \{\ \texttt{branch}\ branch_1\ \{ decl_{1_1} \ldots decl_{1_{m_1}}\ stmt_1\ \} \\ \quad\ \ldots \\ \quad\ \texttt{branch}\ branch_n\ \{ decl_{n_1} \ldots decl_{n_{m_n}}\ stmt_n\ \}\} \end{array}}$$

$$\frac{\begin{array}{c} method\_name \in \{\texttt{REGISTER}, \texttt{REREGISTER}, \texttt{INVITE}, \texttt{REINVITE}, \texttt{CANCEL}, \texttt{BYE}\} \\ \tau \vdash decl_{1_1} \ldots decl_{1_{m_1}} : \tau_1 \qquad \mathsf{response}, \tau_1 \vdash stmt_1 : t_1 \\ \ldots \\ \tau \vdash decl_{n_1} \ldots decl_{n_{m_n}} : \tau_n \qquad \mathsf{response}, \tau_n \vdash stmt_n : t_n \\ t_1 = \ldots = t_n = \mathsf{response} \end{array}}{\begin{array}{l} \tau \vdash direction^? \ \texttt{response}\ method\_name\ \texttt{()} \\ \quad \{\ \texttt{branch}\ branch_1\ \{ decl_{1_1} \ldots decl_{1_{m_1}}\ stmt_1\ \} \\ \quad\ \ldots \\ \quad\ \texttt{branch}\ branch_n\ \{ decl_{n_1} \ldots decl_{n_{m_n}}\ stmt_n\ \}\} \end{array}}$$

The judgments for programs are defined as follows :

$$\frac{\begin{array}{cc} \emptyset \vdash decl_{p_1} \ldots decl_{p_a} : \tau_p & \tau_p \vdash method_{p_1} \ldots \tau_p \vdash method_{p_b} \\ \tau_p \vdash decl_{r_1} \ldots decl_{r_c} : \tau_r & \tau_r \vdash method_{r_1} \ldots \tau_r \vdash method_{r_d} \\ \tau_r \vdash decl_{d_1} \ldots decl_{d_e} : \tau_d & \tau_d \vdash method_{d_1} \ldots \tau_d \vdash method_{d_f} \end{array}}{\begin{array}{l} \vdash \texttt{service}\ service\ \{ \\ \ \texttt{processing}\ \{ decl_{p_1} \ldots decl_{p_a}\ method_{p_1} \ldots method_{p_b} \\ \quad \texttt{registration}\ \{ decl_{r_1} \ldots decl_{r_c}\ method_{r_1} \ldots method_{r_d} \\ \quad\quad \texttt{dialog}\ \{ decl_{d_1} \ldots decl_{d_e}\ method_{d_1} \ldots method_{d_f}\ \}\}\} \end{array}}$$

The type checking rules are, however, not sufficient to ensure that a SPL program is well-formed. Other properties to consider include the following :

- There is at most one incoming and at most one outgoing handler for each method name.
- There is at most one successful `forward` per dialog handler. There can be multiple successful `forwards` for a registration.
- There is something to verify about the branch names. Once a branch is created, it has to be used (or matched by `default`) in every handler that can be executed subsequently. Actually, the semantics doesn't care about this. If the branch is not defined, the message is simply forwarded.

- Assignments to header fields should be checked as to whether the field is assignable.
- Forward cannot appear in a declaration, even the declaration of a local variable.
- Handlers for control methods, *i.e.*, `deploy`, `undeploy`, `unregister`, and `end_dialog`, cannot refer to anything about a message, such as headers or `requestURI`. They cannot use `forward`.
- A handler that has `void` return type must use `forward` in tail position.
- If `forward` succeeds or does not succeed, there are some corresponding constraints on the return value.

B.3 Addresses

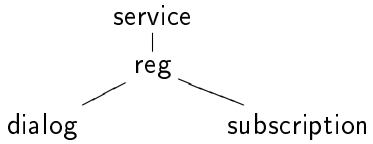
An address is a sequence of unique identifiers defined as follows.

- label = session_type×id, where id is any form of identifier unique within the session_type.
- address = label list

The session types depend on the protocol being addressed. For SPL, we thus have :

- session_type = {service, reg, dialog}

In practice, within a given address, the sequence of session types read from left to right form a non-empty prefix of a path in a tree describing session relationships. For example, if we were to add subscriptions to SPL, the tree would be as follows :



Given such a tree, a valid address is one that contains a prefix of the sequence of session types on a path from the root to a leaf of a tree, where some of the session types are optional. For example, for SPL reg is optional, and thus valid addresses have one of the following forms :

- $\langle(\mathsf{service}, x)\rangle$
- $\langle(\mathsf{service}, x), (\mathsf{reg}, y)\rangle$
- $\langle(\mathsf{service}, x), (\mathsf{reg}, y), (\mathsf{dialog}, z)\rangle$
- $\langle(\mathsf{service}, x), (\mathsf{reg}, y), (\mathsf{subscription}, a)\rangle$
- $\langle(\mathsf{service}, x), (\mathsf{dialog}, z)\rangle$
- $\langle(\mathsf{service}, x), (\mathsf{subscription}, a)\rangle$

The following functions are available for manipulating addresses :

- parent : address → address
- self : address → label
- service : address → label

These functions are defined as follows :

$$\frac{n \geq 1}{\mathsf{parent}(\langle label_1, \ldots, label_n\rangle) = \langle label_1, \ldots, label_{n-1}\rangle}$$

$$\frac{n \geq 1}{\mathsf{self}(\langle label_1, \ldots, label_n\rangle) = label_n}$$

$$\frac{n \geq 1}{\mathsf{service}(\langle label_1, \ldots, label_n\rangle) = label_1}$$

These functions can also be applied to lists of items in one-to-one mapping with a list of addresses, such as the list of associated environments.

B.4 Global State

The global state, σ describes the state of the various sessions and the relationships between them. The state maps an address to a configuration, described as follows :

- $\mathsf{state} = \mathsf{address} \rightarrow \mathsf{branch\ list} \times \mathsf{status} \times (\mathsf{var} \rightarrow \mathsf{value}) \times \mathsf{address\ list}$
- $\mathsf{status} = \mathsf{bool} \times \mathsf{bool} \times \mathsf{string}_\perp \times \mathsf{int}$

status is a tuple composed of four pieces of information : *live*, $persistent_\perp$, *close_fn*, and *open*. *live* indicates whether the session is accepting messages, either its own messages or create session messages of the immediate subsessions. The value is true if this is the case. $persistent_\perp$ indicates whether on ending the session, the data associated with the session persists until the closing of all immediate subsessions. The value is true if this is the case. This field only has a meaningful value if *live* is false, as it is at the moment of ending the session that the semantics chooses whether the ending of the session is persistent. *close_fn* is a string indicating the name of the handler to execute on ending the session, or $\perp$ if there is none. *open* is an integer indicating how many transactions are in progress at the current level. address list is the list of addresses of the subsessions.

There are various kinds of configurations depending on the session being modeled :

- Dialog : In this case, the list of addresses is always empty, *persistent* is false (since a dialog has no subsessions, whether it is persistent or not doesn't matter), and the $\mathsf{string}_\perp$ component has value `end_dialog`.
- Registration : In this case, the list of addresses always contains only addresses of dialogs. The branch list component always contains only one element. The $\mathsf{string}_\perp$ component has value `unregister`.
- Service : In this case, the list of addresses always contains only addresses of registrations. The branch list component always contains one element. The $\mathsf{string}_\perp$ component has value `undeploy`.

If a session has more than one kind of subsession (*i.e.*, if we add subscriptions as subsessions of registrations), then the addresses of these different kinds of subsessions are all mixed together in the address list component. The information about a specific kind of subsession can be obtained by filtering on the session_type in the label at the end of each address.

B.5 Useful Functions

We define the _ notation to ignore a given object field when it is useless in a particular context. The notation $+\!\!+$ concatenates two lists.

We define some functions for accessing and updating the global state σ. The types of these functions are as follows :

- $\mathsf{init\_session} : \mathsf{state} \times \mathsf{address} \times \mathsf{branch\ list} \times \mathsf{bool} \times \mathsf{bool}_\perp \times \mathsf{string}_\perp \times \mathsf{int} \rightarrow \mathsf{state}$
- $\mathsf{lookup\_decls} : \mathsf{program} \times \mathsf{state} \times \mathsf{address} \rightarrow \mathsf{decl\ list}$
- $\mathsf{lookup\_handlers} : \mathsf{program} \times \mathsf{state} \times \mathsf{address} \rightarrow \mathsf{handlers}$
- $\mathsf{lookup\_branches} : \mathsf{state} \times \mathsf{address} \rightarrow \mathsf{branch\ list}$
- $\mathsf{lookup\_live} : \mathsf{state} \times \mathsf{address} \rightarrow \mathsf{bool}$
- $\mathsf{lookup\_persistent} : \mathsf{state} \times \mathsf{address} \rightarrow \mathsf{bool}$

- lookup_close_fn : state × address → string$_\perp$
- lookup_open : state × address → int
- lookup_envs : state × address → (var → value) list
- lookup_dependents : state × address → address list
- update_branches : state × address × branch list → state
- set_persistence : state × address × bool → state
- inc_open : state × address → state
- dec_open : state × address → state
- update_envs : state × address × (var → value) list → state
- delete_config : state × address → state

These functions are defined in terms of the functions get_config : state × address → config and set_config : state × address × config → state, which are defined as follows :

$$\mathsf{get\_config}(\sigma, address) = \sigma(address)$$

$$\mathsf{set\_config}(\sigma, address, config) = \sigma[address \mapsto config]$$

The definitions of the above functions are then as follows :

$$\frac{\begin{array}{c}\mathsf{parent}(address) = \langle\rangle \\ config = \langle branches, \langle live, persistent, close\_fn, open\rangle, \perp, \emptyset\rangle \\ \sigma' = \mathsf{set\_config}(\sigma, address, config)\end{array}}{\mathsf{init\_session}(\sigma, address, branches, live, persistent, close\_fn, open) = \sigma'}$$

$$\frac{\begin{array}{c}\mathsf{parent}(address) \neq \langle\rangle \\ \mathsf{get\_config}(\sigma, \mathsf{parent}(address)) = \langle branches_p, status_p, env_p, subsessions_p\rangle \\ \mathsf{set\_config}(\sigma, \mathsf{parent}(address), \langle branches_p, status_p, env_p, address :: subsessions_p\rangle) = \sigma' \\ config = \langle branches, \langle live, persistent, close\_fn, open\rangle, \perp, \emptyset\rangle \\ \sigma'' = \mathsf{set\_config}(\sigma', address, config)\end{array}}{\mathsf{init\_session}(\sigma, address, branches, live, persistent, close\_fn, open) = \sigma''}$$

$$\frac{\mathsf{self}(address) = \langle session\_type, \_\rangle}{\mathsf{lookup\_decls}(\phi, address) = \phi_{\mathsf{decls}}(\mathsf{service}(address))(session\_type)}$$

$$\mathsf{lookup\_handlers}(\phi, address) = \phi_{\mathsf{handlers}}(\mathsf{service}(address))\rangle$$

$$\frac{\mathsf{get\_config}(\sigma, address) = \langle branches, status, env, subsessions\rangle}{\mathsf{lookup\_branches}(\sigma, address) = branches}$$

$$\frac{\mathsf{get\_config}(\sigma, address) = \langle branches, \langle live, persistent, close\_fn, open\rangle, env, subsessions\rangle}{\mathsf{lookup\_live}(\sigma, address) = live}$$

$$\frac{\mathsf{get\_config}(\sigma, address) = \langle branches, \langle live, persistent, close\_fn, open\rangle, env, subsessions\rangle}{\mathsf{lookup\_persistent}(\sigma, address) = persistent}$$

$$\frac{\mathsf{get\_config}(\sigma, address) = \langle branches, \langle live, persistent, close\_fn, open\rangle, env, subsessions\rangle}{\mathsf{lookup\_close\_fn}(\sigma, address) = close\_fn}$$

$$\frac{\mathsf{get\_config}(\sigma, address) = \langle branches, \langle live, persistent, close\_fn, open\rangle, env, subsessions\rangle}{\mathsf{lookup\_open}(\sigma, address) = open}$$

$$\mathsf{lookup\_envs}(\sigma, \langle\rangle) = \langle\rangle$$

$$\frac{\mathsf{get\_config}(\sigma, address) = \langle \_, \_, env, \_ \rangle \quad \mathsf{lookup\_envs}(\sigma, \mathsf{parent}(address)) = envs}{\mathsf{lookup\_envs}(\sigma, address) = envs +\!\!+ \langle env \rangle}$$

$$\frac{\mathsf{get\_config}(\sigma, address) = \langle branches, status, env, subsessions \rangle}{\mathsf{lookup\_dependents}(\sigma, address) = subsessions}$$

$$\frac{\mathsf{get\_config}(\sigma, address) = \langle \_, status, env, subsessions \rangle \quad \mathsf{set\_config}(\sigma, address, \langle branches, status, env, subsessions \rangle) = \sigma'}{\mathsf{update\_branches}(\sigma, address, branches) = \sigma'}$$

After a call to set_persistence, the *live* attribute is always false. The boolean that is passed to set_persistence is the persistence. There are two rules because it doesn't make sense for something that is not live and not persistent to become persistent. In this case, as treated by the second rule, the persistence is ignored.

$$\frac{\mathsf{get\_config}(\sigma, address) = \langle branches, \langle live, persistent', close\_fn, open \rangle, env, subsessions \rangle \quad live = \mathsf{true} \vee persistent' = \mathsf{true} \quad \mathsf{set\_config}(\sigma, address, \langle branches, \langle \mathsf{false}, persistent, close\_fn, open \rangle, env, subsessions \rangle) = \sigma'}{\mathsf{set\_persistence}(\sigma, address, persistent) = \sigma'}$$

$$\frac{\mathsf{get\_config}(\sigma, address) = \langle branches, \langle live, persistent', close\_fn, open \rangle, env, subsessions \rangle \quad live = \mathsf{false} \wedge persistent' = \mathsf{false}}{\mathsf{set\_persistence}(\sigma, address, persistent) = \sigma}$$

$$\frac{\mathsf{get\_config}(\sigma, address) = \langle branches, \langle live, persistent, close\_fn, open \rangle, env, subsessions \rangle \quad \mathsf{set\_config}(\sigma, address, \langle branches, \langle live, persistent, close\_fn, open + 1 \rangle, env, subsessions \rangle) = \sigma'}{\mathsf{inc\_open}(\sigma, address) = \sigma'}$$

$$\frac{\mathsf{get\_config}(\sigma, address) = \langle branches, \langle live, persistent, close\_fn, open \rangle, env, subsessions \rangle \quad \mathsf{set\_config}(\sigma, address, \langle branches, \langle live, persistent, close\_fn, open - 1 \rangle, env, subsessions \rangle) = \sigma'}{\mathsf{dec\_open}(\sigma, address) = \sigma'}$$

$$\mathsf{update\_envs}(\sigma, \langle \rangle, \langle \rangle) = \sigma$$

$$\frac{\mathsf{get\_config}(\sigma, address) = \langle branches, status, \_, subsessions \rangle \quad \mathsf{set\_config}(\sigma, address, \langle branches, status, \mathsf{self}(envs), subsessions \rangle) = \sigma' \quad \mathsf{update\_envs}(\sigma', \mathsf{parent}(address), \mathsf{parent}(envs)) = \sigma''}{\mathsf{update\_envs}(\sigma, address, envs) = \sigma''}$$

In the second rule for delete_config, we delete the address from the address list of the parent.

$$\frac{\mathsf{parent}(address) = \langle \rangle \quad \mathsf{set\_config}(\sigma, address, \bot) = \sigma'}{\mathsf{delete\_config}(\sigma, address) = \sigma'}$$

$$\frac{\mathsf{parent}(address) \neq \langle \rangle \quad \mathsf{set\_config}(\sigma, address, \bot) = \sigma' \quad \mathsf{get\_config}(\sigma', \mathsf{parent}(address)) = \langle branches, status, env, subsessions \rangle \quad \mathsf{set\_config}(\sigma', \mathsf{parent}(address), \langle branches, status, env, subsessions - address \rangle) = \sigma''}{\mathsf{delete\_config}(\sigma, address, envs) = \sigma''}$$

B.6 Session Management

In terms of the previous functions, we define the functions create_session, prepare_method_invocation, continue_session, and end_session, having the following types :

- create_session : program × state × address × branch list × string$_\perp$ → state
- prepare_method_invocation : program × state × address × direction × method → decl list × stmt × env list × state
- continue_session : program × state × address × env list × branch list → state
- end_session : program × state × address → state

The definitions of these functions are as follows. In the second rule, *handlers* returns the parameter m of the method, and the local declarations *decls* and the code *stmt* associated with the given branch.

$$\frac{\begin{array}{c}\mathsf{lookup\_decls}(\phi, address) = decls \\ \mathsf{init\_session}(\sigma, address, branches, \mathsf{true}, \perp, close\_fn, 0) = \sigma' \\ \mathsf{lookup\_envs}(\sigma', \mathsf{parent}(address)) = parent\_envs \\ \langle parent\_envs, \langle \perp, \emptyset \rangle, \emptyset \rangle \models decls \Rightarrow session\_env \\ \mathsf{update\_envs}(\sigma', address, parent\_envs +\!\!+ \langle session\_env \rangle) = \sigma''\end{array}}{\mathsf{create\_session}(\phi, \sigma, address, branches, close\_fn) = \sigma''}$$

$$\frac{\begin{array}{c}\mathsf{lookup\_handlers}(\phi, address) = handlers \\ \mathsf{lookup\_branches}(\sigma, address) = branches \\ handlers(direction, method, branches) = \langle decls, stmt \rangle \\ \mathsf{lookup\_envs}(\sigma, address) = envs \\ \mathsf{inc\_open}(\sigma, address) = \sigma'\end{array}}{\mathsf{prepare\_method\_invocation}(\phi, \sigma, address, direction, method) = \langle decls, stmt, envs, \sigma' \rangle}$$

$$\frac{\begin{array}{c}\mathsf{update\_envs}(\sigma, address, envs) = \sigma' \\ \mathsf{update\_branches}(\sigma', address, branches) = \sigma'' \\ \mathsf{dec\_open}(\sigma'', address) = \sigma''' \\ \mathsf{lookup\_live}(\sigma''', address) = \mathsf{true}\end{array}}{\mathsf{continue\_session}(\phi, \sigma, address, envs, branches) = \sigma'''}$$

$$\frac{\begin{array}{c}\mathsf{update\_envs}(\sigma, address, envs) = \sigma' \\ \mathsf{update\_branches}(\sigma', address, branches) = \sigma'' \\ \mathsf{dec\_open}(\sigma'', address) = \sigma''' \\ \mathsf{lookup\_live}(\sigma''', address) = \mathsf{false} \\ \mathsf{end\_session}(\phi, \sigma''', address) = \sigma''''\end{array}}{\mathsf{continue\_session}(\phi, \sigma, address, envs, branches) = \sigma''''}$$

All of the remaining rules are used to define end_session. The definition uses the functions end_parents, end_children, and close_session, having the following types :

- end_children : program × state × address → state
- end_parents : program × state × address → state
- close_session : program × state × address → state

Ending a session entails ending all the children, with end_children, and then, if this process is successful, ending any of the parents that are not live. Ending the children entails first moving down the tree of children, setting *live* in each case to false, and then starting from the

leaves, closing each child that has no pending transactions and no dependents. This process is oblivious to whether the children are persistent. Note, however, that when the current level is persistent, this process is only initiated when there are no children, in which case end_children only closes the current level. Ending the parents entails looking for parents that are not live and that if persistent have no dependents. This process works its way up the tree, stopping when reaching a parent that does not have these properties, or when reaching the root of the tree.

close_session invokes the close handler of the current session, once we have decided that the current session should be ended. When there is code to evaluate, it is evaluated with respect to the empty continuation because the code cannot use `forward`.

The following rules decide whether to end the session at the current level. On invoking end_session, the *live* attribute of the current level must be false.

$$\frac{\mathsf{lookup\_persistent}(\sigma, address) = \mathsf{true} \wedge \mathsf{lookup\_dependents}(\sigma, address) \neq \emptyset}{\mathsf{end\_session}(\phi, \sigma, address) = \sigma}$$

$$\frac{\begin{array}{c}\mathsf{lookup\_persistent}(\sigma, address) = \mathsf{false} \vee \mathsf{lookup\_dependents}(\sigma, address) = \emptyset \\ \mathsf{end\_children}(\phi, \sigma, address) = \sigma' \\ \mathsf{end\_parents}(\phi, \sigma', \mathsf{parent}(address)) = \sigma''\end{array}}{\mathsf{end\_session}(\phi, \sigma, address) = \sigma''}$$

The following rules end the session and all of its children, starting at the leaves. These rules assume that σ is up to date with respect to the various session environments. end_children' of type program × state × address × label list → state is used to loop over the list of dependents.

$$\frac{\begin{array}{c}\mathsf{lookup\_dependents}(\sigma, address) = \emptyset \wedge \mathsf{lookup\_open}(\sigma, address) = 0 \\ \mathsf{close\_session}(\phi, \sigma, address) = \sigma'\end{array}}{\mathsf{end\_children}'(\phi, \sigma, address, \langle\rangle) = \sigma'}$$

$$\frac{\mathsf{lookup\_dependents}(\sigma, address) \neq \emptyset \vee \mathsf{lookup\_open}(\sigma, address) > 0}{\mathsf{end\_children}'(\phi, \sigma, address, \langle\rangle) = \sigma}$$

$$\frac{\begin{array}{c}\mathsf{set\_persistence}(\sigma, child\_address, \mathsf{false}) = \sigma' \\ \mathsf{end\_children}(\phi, \sigma', child\_address) = \sigma'' \\ \mathsf{end\_children}'(\phi, \sigma'', address, \pi) = \sigma'''\end{array}}{\mathsf{end\_children}'(\phi, \sigma, address, child\_address :: \pi) = \sigma'''}$$

$$\frac{\mathsf{end\_children}'(\phi, \sigma, address, \mathsf{lookup\_dependents}(\sigma, address)) = \sigma'}{\mathsf{end\_children}(\phi, \sigma, address) = \sigma'}$$

The following rules end any of the parents that are not live and now have no more subsessions. These rules assume that σ is up to date with respect to the various session environments.

$$\mathsf{end\_parents}(\phi, \sigma, \langle\rangle) = \sigma$$

$$\frac{\mathsf{lookup\_live}(\sigma, address) = \mathsf{true} \vee \mathsf{lookup\_dependents}(\sigma, address) \neq \emptyset}{\mathsf{end\_parents}(\phi, \sigma, address) = \sigma}$$

$$\frac{\begin{array}{c}\mathsf{lookup\_live}(\sigma, address) = \mathsf{false} \wedge \mathsf{lookup\_dependents}(\sigma, address) = \emptyset \\ \mathsf{close\_session}(\phi, \sigma, address) = \sigma' \\ \mathsf{end\_parents}(\phi, \sigma', \mathsf{parent}(address)) = \sigma''\end{array}}{\mathsf{end\_parents}(\phi, \sigma, address) = \sigma''}$$

The following rules perform the actions required locally to close a single session. These rules assume that σ is up to date with respect to the various session environments.

$$\frac{\begin{array}{c}\mathsf{lookup\_close\_fn}(\sigma, address) = \bot \\ \mathsf{delete\_config}(\sigma, address) = \sigma'\end{array}}{\mathsf{close\_session}(\phi, \sigma, address) = \sigma'}$$

$$\frac{\begin{array}{c}\mathsf{lookup\_close\_fn}(\sigma, address) = close\_fn \\ \mathsf{prepare\_method\_invocation}(\phi, \sigma, address, \bot, close\_fn) = \langle decls, stmt, envs, \sigma' \rangle \\ \tau = \langle \sigma, address \rangle \qquad r = \langle envs, \langle \bot, \langle \bot, \emptyset \rangle \rangle \rangle \\ \tau, r, \langle \texttt{CLOSE} \rangle \models decls, stmt \Rightarrow^* \_, \sigma' \\ \mathsf{delete\_config}(\sigma', address) = \sigma''\end{array}}{\mathsf{close\_session}(\phi, \sigma, address) = \sigma''}$$

$$\frac{\mathsf{update\_envs}(\sigma, address, \rho_{\mathsf{envs}}) = \sigma'}{\langle \sigma, address \rangle, \rho \models \langle \texttt{CLOSE} \rangle, \_, \_ \Rightarrow \bot, \sigma'}$$

B.7 Core Language Constructs

In this section, we present some useful environments which are manipulated by the semantic rules. Then, we present the semantic rule for each kind of language constructs : declarations, handlers, statements and expression.

B.7.1 Environments

The semantics uses the following environments :

- *envs* : A sequence of environments corresponding to the current address. For SPL, this contains at most the following environments :
  - *env* : The service environment.
  - *rid_ reg_ env* : The register environment for a registration instance.
  - *did_ dialog_ env* : The dialog environment for a dialog instance.
- *message_ info* : A pair of the requestURI and an environment containing bindings derived from the message headers.
- *local_ env* : The local environment.

All of these environments (except *envs*) are mappings from variables to values. The semantics refers to these environments collectively as ρ which is a tuple of the above environments in the above order. The individual environments are accessed by *e.g.* $\rho_{\mathsf{message\_info}}$.

In the evaluation of the entry point of a handler, the environment r is created, which is the same as ρ, but does not contain a local environment.

B.7.2 Judgments for Declarations, Statements and Expressions

The semantics is organized as an abstract machine containing the following kinds of configurations :

- Method invocation : $message, \phi, \sigma \models_{\text{mi}} service$

  A *message* has the form $\langle method, direction, headers, rq \rangle$, where rq is the request URI of the message.
- Handler body : $\tau, r, s \models_{\text{h}} decls, stmt$

  τ gives context information and has the form $\langle \sigma, address \rangle$. This information is not used in the normal course of executing the body of a handler, but is used in the treatment of (`parallel`) `forward`.
  r is the top level environment composed of hierarchical session environment and message environment (request URI and headers).
  s is the stack of continuations.
- Declaratrion : $\rho \models_{\text{d}} decls : env$

  ρ is similiar to r but also include the local environment of the invoked method.
- Statement : $\tau, \rho, s \models_{\text{s}} stmt$
- Statement continuation : $\tau, \rho \models_{\text{sc}} s, resp_{\perp}, branch_{\perp}$

  $resp_{\perp}$ is either a response or $\perp$. $\perp$ is used for statements other than `return`, which do not have a response. $branch_{\perp}$ is defined similarly.
- Expression : $\tau, \rho, s \models_{\text{e}} exp$
- Expression continuation : $\tau, \rho \models_{\text{ec}} s, value$

  value is an arbitrary value, which is not necessarily a response.
- SIP : A SIP configuration is a call to a function of the underlying platform, such as forward.
- Terminal : $resp_{\perp}, \sigma$

The execution of a handler is represented by a sequence of configurations, related according to the semantic rules. The initial configuration is a service configuration. The final configuration is a terminal configuration. A configuration between the initial and final configurations is any kind of configuration other than a service configuration or a terminal configuration.

The semantics is essentially a continuation-based semantics, but where the continuations have been defunctionalized as abstract machine instructions. We introduce the various instructions as they are needed.

In general, a semantic rule has the form :

$$\frac{\begin{array}{c} precondition_1 \\ \vdots \\ precondition_n \end{array}}{configuration \Rightarrow configuration'}$$

$precondition_1 \ldots precondition_n$ describe operations that must be performed given the information provided in *configuration* in order to produce $configuration'$.

B.7.3 Entry Point of the Semantics

The first step of the semantics is to identify the handler to execute.

Service Methods

The only service methods are `deploy`, which is used to create a service, and `undeploy`, which is used to end a service.

The following rules define the semantics of `deploy`, which is assumed like all control methods, to succeed. There is one continuation `DEPLOY` ϕ.

$$\frac{\begin{array}{c} address = \langle service \rangle \\ \mathsf{create\_session}(\phi, \sigma, address, \langle \texttt{default} \rangle, \texttt{undeploy}) = \sigma' \\ \mathsf{prepare\_method\_invocation}(\phi, \sigma', address, \bot, \texttt{deploy}) = \langle decls, stmt, envs, \sigma'' \rangle \\ \tau = \langle \sigma'', address \rangle \qquad r = \langle envs, \langle \bot, \emptyset \rangle \rangle \end{array}}{\langle \mathsf{deploy}, \bot, \bot, \bot \rangle, \phi, \sigma \models_{\text{mi}} service \Rightarrow \tau, r, \langle \texttt{DEPLOY} \ \ \phi \rangle \models_{\text{h}} decls, stmt}$$

$$\frac{\mathsf{continue\_session}(\phi, \sigma, address, \rho_{\mathsf{envs}}, \langle b \rangle) = \sigma'}{\langle \sigma, address \rangle, \rho, \models_{\text{sc}} \langle \texttt{DEPLOY} \ \ \phi \rangle, \texttt{SUCCESS}, b \Rightarrow \bot, \sigma'}$$

$$\frac{\begin{array}{c} \mathsf{set\_persistence}(\sigma, address, \mathsf{false}) = \sigma' \\ \mathsf{continue\_session}(\phi, \sigma', address, \rho_{\mathsf{envs}}, \langle b \rangle) = \sigma'' \end{array}}{\langle \sigma, address \rangle, \rho, \models_{\text{sc}} \langle \texttt{DEPLOY} \ \ \phi \rangle, \texttt{ERROR}, b \Rightarrow \bot, \sigma''}$$

The semantics of `undeploy` simply ends the session. Because a service is not persistent, this ends all subsessions as well.

$$\frac{\begin{array}{c} address = \langle service \rangle \\ \mathsf{set\_persistence}(\sigma, address, persistent) = \sigma' \\ \mathsf{end\_session}(\phi, \sigma', address) = \sigma'' \end{array}}{\langle \mathsf{undeploy}(persistent), \bot, \bot, \bot \rangle, \phi, \sigma \models_{\text{mi}} service \Rightarrow \bot, \sigma''}$$

Registration Methods

The only registration methods are `initial REGISTER`, which is used to create a registration, and `medial REGISTER`, which is used to keep the registration alive. An initial `REGISTER` creates an entry for a new registration id, initializes the registration variables, and sets the live bit associated with the registration entry to true. The initial values of the registration variables cannot refer to the branch lists of the message (headers or request URI). A medial `REGISTER` can update the variables associated with the registration.

The following three rules define the semantics of an initial `REGISTER`. There is one continuation : `INITIAL_REG` ϕ.

$$\frac{\begin{array}{c} address = \langle service, rid \rangle \\ \mathsf{create\_session}(\phi, \sigma, address, \langle \texttt{default} \rangle, \texttt{unregister}) = \sigma' \\ \mathsf{prepare\_method\_invocation}(\phi, \sigma', address, direction, \texttt{REGISTER}) = \langle decls, stmt, envs, \sigma'' \rangle \\ \tau = \langle \sigma'', address \rangle \qquad r = \langle envs, \langle rq, headers \rangle \rangle \end{array}}{\begin{array}{l} \langle \texttt{REGISTER}(rid), direction, rq, headers \rangle, \phi, \sigma \models_{\text{mi}} service \\ \qquad \Rightarrow \tau, r, \langle \texttt{INITIAL\_REG} \ \ \phi \rangle \models_{\text{h}} decls, stmt \end{array}}$$

There are two rules for the continuation `INITIAL_REG` ϕ : one where the registration has succeeded and one where it has failed.

$$\frac{\mathsf{continue\_session}(\phi, \sigma, address, \rho_{\mathsf{envs}}, \langle b \rangle) = \sigma'}{\langle \sigma, address \rangle, \rho \models_{\mathrm{sc}} \langle \texttt{INITIAL\_REG}\ \phi \rangle, \texttt{/SUCCESS/}resp(resp\_headers), b \Rightarrow \texttt{/SUCCESS/}resp(resp\_headers), \sigma'}$$

$$\frac{\begin{array}{c}\mathsf{set\_persistence}(\sigma, address, \mathsf{false}) = \sigma' \\ \mathsf{continue\_session}(\phi, \sigma', address, \rho_{\mathsf{envs}}, \langle b \rangle) = \sigma''\end{array}}{\langle \sigma, address \rangle, \rho \models_{\mathrm{sc}} \langle \texttt{INITIAL\_REG}\ \phi \rangle, \texttt{/ERROR/}resp(resp\_headers), b \Rightarrow \texttt{/ERROR/}resp(resp\_headers), \sigma''}$$

The following rule is for a medial `REGISTER`, *i.e.* `REREGISTER`. There is one continuation in this case : `MEDIAL_REG` ϕ.

$$\frac{\begin{array}{c}address = \langle service, rid \rangle \\ \mathsf{prepare\_method\_invocation}(\phi, \sigma, address, direction, \texttt{REREGISTER}) = \langle decls, stmt, envs, \sigma' \rangle \\ \tau = \langle \sigma', address \rangle \qquad r = \langle envs, \langle rq, headers \rangle \rangle\end{array}}{\langle \texttt{REREGISTER}(rid), direction, rq, headers \rangle, \phi, \sigma \models_{\mathrm{mi}} service \Rightarrow \tau, r, \langle \texttt{MEDIAL\_REG}\ \phi \rangle \models_{\mathrm{h}} decls, stmt}$$

$$\frac{\mathsf{continue\_session}(\phi, \sigma, address, \rho_{\mathsf{envs}}, \langle b \rangle) = \sigma'}{\langle \sigma, address \rangle, \rho \models_{\mathrm{sc}} \langle \texttt{MEDIAL\_REG}\ \phi \rangle, resp, b \Rightarrow resp, \sigma'}$$

The lifetime of a registration is limited by a header associated with the `REGISTER` method (either initial or medial) or a default value. At the time of this expiration, the support for SPL in the underlying platform generates a $\mathsf{registration\_timeout}(service,\ rid,\ \phi,\ \sigma)$ message. This message sets the *live* field of the registration to false. If there are not more sessions associated with the registration, then the registration is deleted. If there are still sessions associated with the registration, then the registration is deleted on the completion of a subsequent `BYE` method when there are no remaining live sessions. All of this is taken care of by $\mathsf{end\_session}$, because a registration is declared to be persistent.

$$\frac{\begin{array}{c}address = \langle service, rid \rangle \\ \mathsf{set\_persistence}(\sigma, address, \mathsf{true}) = \sigma' \\ \mathsf{end\_session}(\phi, \sigma', address) = \sigma''\end{array}}{\mathsf{registration\_timeout}(service,\ rid,\ \phi,\ \sigma) \Rightarrow \bot, \sigma''}$$

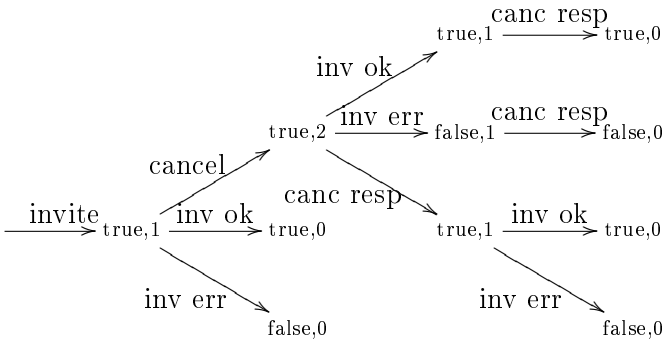
$$\frac{\begin{array}{c}address = \langle service, rid \rangle \\ \mathsf{set\_persistence}(\sigma, address, persistent) = \sigma' \\ \mathsf{end\_session}(\phi, \sigma', address) = \sigma''\end{array}}{\mathsf{unregister}(service,\ rid,\ \phi,\ \sigma, persistent) \Rightarrow \bot, \sigma''}$$

Dialog Methods

A dialog is initiated by an `INVITE`. As for `REGISTER`, there are two kinds of `INVITE` : initial and medial. There is no final `INVITE`, because that is implemented as `BYE`. If the service includes a `registration` block, then `INVITE` will only been invoked when `REGISTER` has previously been invoked, and the call to `INVITE` will contain information about the associated *rid*. If the service does not include a `registration` block, then the method has only *did* as an argument, and the address is constructed as $\langle service, did \rangle$. This strategy applies to the other dialog methods as well. The semantics of `initial INVITE` is defined by the following rules. There is one continuation : `INITIAL_INVITE` *direction handlers*.

$$\frac{\begin{array}{c} address = \langle service, rid, did \rangle \\ \mathsf{lookup\_branches}(\sigma, \mathsf{parent}(address)) = \langle branch \rangle \\ \mathsf{create\_session}(\phi, \sigma, address, \langle branch \rangle, \texttt{undialog}) = \sigma' \\ \mathsf{prepare\_method\_invocation}(\phi, \sigma', direction, \texttt{INVITE}) = \langle decls, stmt, envs, \sigma'' \rangle \\ \tau = \langle \sigma'', address \rangle \qquad r = \langle envs, \langle rq, headers \rangle \rangle \end{array}}{\begin{array}{r} \langle \texttt{INVITE}(rid, did), direction, rq, headers \rangle, \phi, \sigma \models_{\text{mi}} service \\ \Rightarrow \tau, r, \langle \texttt{INITIAL\_INVITE}\ \ \phi \rangle \models_{\text{h}} decls, stmt \end{array}}$$

Because we do not control the ordering in which messages are transmitted to the service, there are a number of three possibilities, as shown by the following diagram. At each node, the boolean value indicates whether the dialog is live and the integer value indicates the number of open transactions. At any point where the node contains false and 0, the session will be ended when by `continue_session`.



The rules are as follows :

$$\frac{\begin{array}{c} \mathsf{lookup\_branches}(\sigma, address) = branches \\ \mathsf{continue\_session}(\phi, \sigma, address, \rho_{\mathsf{envs}}, \mathsf{add}(b, branches)) = \sigma' \end{array}}{\begin{array}{r} \langle \sigma, address \rangle, \rho \models_{\text{sc}} \langle \texttt{INITIAL\_INVITE}\ \ \phi \rangle, \texttt{/SUCCESS/}resp\,\texttt{(}resp\_headers\texttt{)}, b \\ \Rightarrow \texttt{/SUCCESS/}resp\,\texttt{(}resp\_headers\texttt{)}, \sigma' \end{array}}$$

The function $\mathsf{add}(b, branches)$ adds b at the beginning of *branches* if $b \notin branches$ and returns *branches* otherwise.

$$\frac{\begin{array}{c}\mathsf{set\_persistence}(\sigma, address, \mathsf{false}) = \sigma' \\ \mathsf{lookup\_branches}(\sigma', address) = branches \\ \mathsf{continue\_session}(\phi, \sigma', address, \rho_{\mathsf{envs}}, \mathsf{add}(b, branches)) = \sigma''\end{array}}{\langle \sigma, address\rangle, \rho \models_{\text{sc}} \langle \texttt{INITIAL\_INVITE}\ \ \phi\rangle, /\texttt{ERROR}/resp\,(resp\_headers)\,, b \Rightarrow /\texttt{ERROR}/resp\,(resp\_headers)\,, \sigma''}$$

The following rule is for methods that occur within an existing dialog. The method can be ACK, CANCEL or medial INVITE. There is one continuation : MEDIAL_INVITE ϕ.

$$\frac{\begin{array}{c}address = \langle service, rid, did\rangle \\ \mathsf{prepare\_method\_invocation}(\phi, \sigma, address, direction, \texttt{REINVITE}) = \langle decls, stmt, envs, \sigma'\rangle \\ \tau = \langle \sigma', address\rangle \qquad r = \langle envs, \langle rq, headers\rangle\rangle\end{array}}{\langle \texttt{REINVITE}(rid, did), direction, rq, headers\rangle, \phi, \sigma \models_{\text{mi}} service \Rightarrow \tau, r, \langle \texttt{MEDIAL\_INVITE}\ \ \phi\rangle \models_{\text{h}} decls, stmt}$$

$$\frac{\begin{array}{c}\mathsf{lookup\_branches}(\sigma, address) = branches \\ \mathsf{continue\_session}(\phi, \sigma, address, \rho_{\mathsf{envs}}, \mathsf{add}(b, branches)) = \sigma'\end{array}}{\langle \sigma, address\rangle, \rho \models_{\text{sc}} \langle \texttt{MEDIAL\_INVITE}\ \ \phi\rangle, resp, b \Rightarrow resp, \sigma'}$$

The following rules are for BYE, which ends a dialog. If the registration is still live or there are other dialogs associated with the registration, then we simply delete the dialog from the dialog environment. If the registration is no longer live and there are no other dialogs associated with the registration, then we run the unregister code and delete the entire registration (including the dialog). There is one continuation : BYE ϕ.

$$\frac{\begin{array}{c}address = \langle service, rid, did\rangle \\ \mathsf{prepare\_method\_invocation}(\phi, \sigma, address, direction, \texttt{BYE}) = \langle decls, stmt, envs, \sigma'\rangle \\ \tau = \langle \sigma', address\rangle \qquad r = \langle envs, \langle rq, headers\rangle\rangle\end{array}}{\langle \texttt{BYE}(rid, did), direction, rqheaders\rangle, \phi, \sigma \models_{\text{mi}} service \Rightarrow \tau, r, \langle \texttt{BYE}\ \ \phi\rangle \models_{\text{h}} decls, stmt}$$

If the registration has not ended or the registration has ended and there are dialogs other than the current one, then the registration is preserved and we only end the dialog. Otherwise we end both the dialog and the enclosing registration. This is taken care of end_session, because registration is persistent.

$$\frac{\begin{array}{c}\mathsf{lookup\_branches}(\sigma, address) = branches \\ \mathsf{set\_persistence}(\sigma, address, \mathsf{false}) = \sigma' \\ \mathsf{continue\_session}(\phi, \sigma', address, \rho_{\mathsf{envs}}, \mathsf{add}(b, branches)) = \sigma''\end{array}}{\langle \sigma, address\rangle, \rho \models_{\text{sc}} \langle \texttt{BYE}\ \ \phi\rangle, resp, b \Rightarrow resp, \sigma''}$$

The following only occurs when no ACK appears after an INVITE.

$$\frac{\begin{array}{c}address = \langle service, rid, did\rangle \\ \mathsf{set\_persistence}(\sigma, address, \mathsf{false}) = \sigma' \\ \mathsf{end\_session}(\phi, \sigma', address) = \sigma''\end{array}}{\mathsf{dialog\_timeout}(service,\ rid,\ did,\ \phi,\ \sigma) \Rightarrow \bot, \sigma''}$$

Default Handling

When there is no handler, and thus nothing to do, the semantics just forwards the message as normal. Actually, the semantics does not distinguish between the case where there is no handler and the case where there is a handler but there is no branch. The latter case should never occur.

The following rule is for an `REGISTER` :

$$\frac{\begin{array}{c} address = \langle service, rid \rangle \\ \mathsf{create\_session}(\phi, \sigma, address, \langle \texttt{default} \rangle, \texttt{unregister}) = \sigma' \\ \mathsf{prepare\_method\_invocation}(\phi, \sigma', address, direction, \texttt{REGISTER}) = \langle \bot, \bot, envs, \sigma'' \rangle \\ \tau = \langle \sigma'', address \rangle \qquad \rho = \langle envs, \langle rq, headers \rangle \rangle \end{array}}{\begin{array}{l} \langle \texttt{REGISTER}(rid), direction, rq, headers \rangle, \phi, \sigma \models_{\text{mi}} service \\ \Rightarrow \tau, \rho, \langle \texttt{INITIAL\_REG } \phi \rangle \models_{\text{h}} \langle \rangle, \texttt{return forward branch default} \end{array}}$$

The rules for the other methods are similar, *i.e.*, we pretend that the statement is `return forward branch default` and then use the original continuation.

B.7.4 Semantics of Handler Bodies

A handler can only declare local variables at the root of the body. The rules here evaluate such declarations and then continue with the semantics of statements. Note that even the variables associated with `when` must be declared at the top level, and thus `when` is implemented by assignments. This implies that all `when`-declared variables (like all other local declarations) have to be given unique names.

$$\frac{\begin{array}{c} r = \langle envs, message\_info \rangle \\ \langle envs, message\_info, \emptyset \rangle \models_{\text{d}} decls : local\_env \\ \rho = \langle envs, message\_info, local\_env \rangle \end{array}}{\tau, r, s \models_{\text{h}} decls, stmt \Rightarrow \tau, \rho, s \models_{\text{s}} stmt}$$

B.7.5 Semantics of Statements

Sequence

We assume that a sequence has one of the following forms :

- $\{\,\}$
- $\{\, stmt_1 \;\; stmt_2 \,\}$

These forms are not particularly convenient for programming, but it is easy to turn any sequence statement into one of these forms, either by adding braces or by dropping braces that only enclose one statement.

$$\tau, \rho, s \models_{\text{s}} \{\,\} \Rightarrow \tau, \rho \models_{\text{sc}} s, \bot, \bot$$

In the case of a sequence of two statements, there is one continuation : `SEQ` $stmt_2$.

$$\tau, \rho, s \models_{\text{s}} \{\, stmt_1 \;\; stmt_2 \,\} \Rightarrow \tau, \rho, (\texttt{SEQ} \;\; stmt_2) :: s \models_{\text{s}} stmt_1$$

$$\tau, \rho \models_{\text{sc}} (\texttt{SEQ} \;\; stmt_2) :: s, \_, \_ \Rightarrow \tau, \rho, s \models_{\text{s}} stmt_2$$

Assignment

Note that identifiers and SIP headers are unified ; a SIP header is just an identifier that is found in *headers*.

In the case of an assignment, there is one continuation : `ASSIGN` x.

$$\tau, \rho, s \models_{\text{s}} x \;=\; exp \Rightarrow \tau, \rho, (\texttt{ASSIGN} \;\; x) :: s \models_{\text{e}} exp$$

$$\tau, \rho \models_{\text{ec}} (\texttt{ASSIGN} \;\; x) :: s, v \Rightarrow \tau, \mathsf{update}(x, v, \rho) \models_{\text{sc}} s, \bot, \bot$$

$\mathsf{update}(x, v, \langle envs, \langle rq, headers\rangle, local\_\ env\rangle) = \mathsf{update}_l(x, v, envs, \langle rq, headers\rangle, local\_\ env)$, which is defined as follows :

$$\frac{x \in \mathsf{dom}(local\_\ env)}{\mathsf{update}_l(x, v, envs, message\_\ info, local\_\ env) = \langle envs, message\_\ info, local\_\ env[x \mapsto v]\rangle}$$

$$\frac{x \notin \mathsf{dom}(local\_\ env) \qquad \mathsf{update}_h(x, v, envs, message\_\ info) = \langle envs', message\_\ info'\rangle}{\mathsf{update}_l(x, v, envs, message\_\ info, local\_\ env) = \langle envs', message\_\ info', local\_\ env\rangle}$$

$$\frac{x \in \mathsf{dom}(headers)}{\mathsf{update}_h(x, v, envs, \langle rq, headers\rangle) = \langle envs, \langle rq, headers[x \mapsto v]\rangle\rangle}$$

$$\frac{x \notin \mathsf{dom}(headers) \qquad \mathsf{update}_e(x, v, envs) = envs'}{\mathsf{update}_h(x, v, envs, \langle rq, headers\rangle) = \langle envs', \langle rq, headers\rangle\rangle}$$

$$\frac{x \in \mathsf{dom}(env)}{\mathsf{update}_e(x, v, envs +\!\!+ \langle env\rangle) = envs +\!\!+ \langle env[x \mapsto v]\rangle}$$

$$\frac{x \notin \mathsf{dom}(env) \qquad \mathsf{update}_e(x, v, envs) = envs'}{\mathsf{update}_e(x, v, envs +\!\!+ \langle env\rangle) = envs' +\!\!+ \langle env\rangle}$$

Return

Return is associated with one continuation : `RETURN` b.

$$\tau, \rho, s \models_{\text{s}} \texttt{return} \;\; exp \;\; \texttt{branch} \;\; b\,\texttt{;} \Rightarrow \tau, \rho, (\texttt{RETURN} \;\; b) :: s \models_{\text{e}} exp$$

$$\tau, \rho \models_{\text{ec}} (\texttt{RETURN} \;\; b) :: s, resp \Rightarrow \tau, \rho \models_{\text{sc}} s, resp, b$$

The following returns no value but changes the branch.

$$\tau, \rho, s \models_{\text{s}} \texttt{return branch } b\texttt{;} \Rightarrow \tau, \rho \models_{\text{sc}} s, \bot, b$$

If

If is associated with one continuation : `IF` $stmt_1$ $stmt_2$.

$$\tau, \rho, s \models_{\text{s}} \texttt{if (}exp\texttt{)} \ stmt_1 \ \texttt{else} \ stmt_2 \Rightarrow \tau, \rho, (\texttt{IF} \ stmt_1 \ stmt_2) :: s \models_{\text{e}} exp$$

$$\tau, \rho \models_{\text{ec}} (\texttt{IF} \ stmt_1 \ stmt_2) :: s, \mathsf{true} \Rightarrow \tau, \rho, s \models_{\text{s}} stmt_1$$

$$\tau, \rho \models_{\text{ec}} (\texttt{IF} \ stmt_1 \ stmt_2) :: s, \mathsf{false} \Rightarrow \tau, \rho, s \models_{\text{s}} stmt_2$$

When

$$\begin{aligned} \tau, \rho, s \models_{\text{s}} &\texttt{when} \ exp \ \texttt{(}hd_1 \ t_1 \ x_1\texttt{,...,} \ hd_n \ t_n \ x_n\texttt{)} \ stmt_1 \ \texttt{else} \ stmt_2 \\ &\Rightarrow \tau, \rho, (\texttt{WHEN} \ \langle hd_1 \ t_1 \ x_1, \ldots, hd_n \ t_n \ x_n \rangle \ stmt_1 \ stmt_2) :: s \models_{\text{e}} exp \end{aligned}$$

$$\frac{\begin{array}{c}\{hd_1, \ldots, hd_n\} \subseteq \mathsf{dom}(headers) \\ \rho' = \mathsf{update}(x_1, \mathsf{coerce}(headers(hd_1), t_1), \ldots \mathsf{update}(x_n, \mathsf{coerce}(headers(hd_n), t_n), \rho) \ldots)\end{array}}{\tau, \rho \models_{\text{ec}} (\texttt{WHEN} \ \langle hd_1 \ t_1 \ x_1, \ldots, hd_n \ t_n \ x_n \rangle \ stmt_1 \ stmt_2) :: s, \langle rq, headers \rangle \Rightarrow \tau, \rho', s \models_{\text{s}} stmt_1}$$

$$\frac{\{hd_1, \ldots, hd_n\} \not\subseteq \mathsf{dom}(headers)}{\tau, \rho \models_{\text{ec}} (\texttt{WHEN} \ \langle hd_1 \ t_1 \ x_1, \ldots, hd_n \ t_n \ x_n \rangle \ stmt_1 \ stmt_2) :: s, \langle rq, headers \rangle \Rightarrow \tau, \rho, s \models_{\text{s}} stmt_2}$$

$$\frac{\begin{array}{c}\{hd_1, \ldots, hd_n\} \subseteq \mathsf{dom}(resp\_headers) \\ \rho' = \mathsf{update}(x_1, \mathsf{coerce}(resp\_headers(hd_1), t_1), \ldots \\ \mathsf{update}(x_n, \mathsf{coerce}(resp\_headers(hd_n), t_n), \rho) \ldots)\end{array}}{\begin{array}{r}\tau, \rho \models_{\text{ec}} (\texttt{WHEN} \ \langle hd_1 \ t_1 \ x_1, \ldots, hd_n \ t_n \ x_n \rangle \ stmt_1 \ stmt_2) :: s, code(resp\_headers) \\ \Rightarrow \tau, \rho', s \models_{\text{s}} stmt_1\end{array}}$$

$$\frac{\{hd_1, \ldots, hd_n\} \not\subseteq \mathsf{dom}(resp\_headers)}{\begin{array}{r}\tau, \rho \models_{\text{ec}} (\texttt{WHEN} \ \langle hd_1 \ t_1 \ x_1, \ldots, hd_n \ t_n \ x_n \rangle \ stmt_1 \ stmt_2) :: s, code(resp\_headers) \\ \Rightarrow \tau, \rho, s \models_{\text{s}} stmt_2\end{array}}$$

With

We assume there is only one binding, as multiple bindings can be implemented by a sequence of `with`s. There are two continuations : `WITH1` *field* exp_2 and `WITH2` *field resp*. This statement could only be used to update a request with the `this` expression.

$$\tau, \rho, s \models_{s} exp_1 \texttt{ with } field = exp_2 \texttt{ ;} \Rightarrow \tau, \rho, (\texttt{WITH1 } field \ exp_2) :: s \models_{e} exp_1$$

$$\tau, \rho \models_{ec} (\texttt{WITH1 } field \ exp_2) :: s, v \Rightarrow \tau, \rho, (\texttt{WITH2 } field \ v) :: s \models_{e} exp_2$$

$$\frac{\rho = \langle envs, \langle rq, headers \rangle, local\_env \rangle \quad \langle envs, \langle rq, headers[field \mapsto v] \rangle, local\_env \rangle = \rho'}{\tau, \rho \models_{ec} (\texttt{WITH2 } field \ \langle rq, headers \rangle) :: s, v \Rightarrow \tau, \rho' \models_{sc} s, \bot, \bot}$$

B.7.6 Semantics of Expressions

We consider a simplified set of expressions in which there are no `with` expressions, identifiers and SIP headers are unified as described above for assignments, the only binary operator is ==, there are no unary operators, there are no `pop` expressions, and there are no function calls. An expression and a `branch` are obligatory in every `(parallel) forward`. The expression must evaluate to a list. We add an expression for making a list out of the value of a single expression so that the value of `requestURI` can be put into a list to make a suitable argument for the trivial case.

An expression can reinvoke the underlying machine, via a `forward` or `parallel forward`. For this, we have to bring σ up to date with respect to the changes that have occurred in *envs*. When control returns to the semantics, we similarly have to obtain the new *envs*. For these operations, we have to keep the environments separated. The semantics of expressions also needs to know σ, and the current *address*.

Forward

$$\frac{\mathsf{update\_envs}(\sigma, address, \rho_{\mathsf{envs}}) = \sigma' \quad \rho_{\mathsf{message\_info}} = \langle m, \langle rq, headers \rangle \rangle}{\langle \sigma, address \rangle, \rho, s \models_{e} \texttt{forward} \Rightarrow \mathsf{forward}(rq, false, (\texttt{FORWARD } address \ \rho_{\mathsf{message\_info}} \ \rho_{\mathsf{local\_env}}) :: s, headers, \sigma')}$$

$$\tau, \rho, s \models_{e} \texttt{forward } exp \Rightarrow \tau, \rho, \texttt{FORWARD\_EXP} :: s \models_{e} exp$$

$$\frac{\mathsf{update\_envs}(\sigma, address, \rho_{\mathsf{envs}}) = \sigma' \quad \rho_{\mathsf{message\_info}} = \langle m, \langle rq, headers \rangle \rangle}{\langle \sigma, address \rangle, \rho \models_{ec} \texttt{FORWARD\_EXP} :: s, v \Rightarrow \mathsf{forward}(v, false, (\texttt{FORWARD } address \ \rho_{\mathsf{message\_info}} \ \rho_{\mathsf{local\_env}}) :: s, headers, \sigma')}$$

$$\frac{\begin{array}{c}\mathsf{lookup\_envs}(\sigma, address) = envs \\ \tau = \langle \sigma, address \rangle \qquad \rho = \langle envs, message\_info, local\_env \rangle\end{array}}{\begin{array}{l}\mathsf{forward\_response}(resp, (\texttt{FORWARD}\ \ address\ \ message\_info\ \ local\_env) :: s, \sigma) \\ \qquad \Rightarrow \tau, \rho \models_{\text{ec}} s, resp\end{array}}$$

Parallel Forward

$$\frac{\rho_{\mathsf{message\_info}} = \langle m, \langle rq, headers \rangle \rangle}{\begin{array}{l}\tau, \rho, s \models_{\text{e}} \texttt{parallel forward} \\ \Rightarrow \mathsf{forward}(rq, true, (\texttt{FORWARD}\ \ address\ \ \rho_{\mathsf{message\_info}}\ \ \rho_{\mathsf{local\_env}}) :: s, headers, \sigma')\end{array}}$$

$$\tau, \rho, s \models_{\text{e}} \texttt{parallel forward}\ \ exp \Rightarrow \tau, \rho, \texttt{PFORWARD} :: s \models_{\text{e}} exp$$

$$\frac{\begin{array}{c}\mathsf{update\_envs}(\sigma, address, \rho_{\mathsf{envs}}) = \sigma' \\ \rho_{\mathsf{message\_info}} = \langle m, \langle rq, headers \rangle \rangle\end{array}}{\begin{array}{l}\langle \sigma, address \rangle, \rho \models_{\text{ec}} \texttt{PFORWARD} :: s, v \\ \Rightarrow \mathsf{forward}(v, true, (\texttt{FORWARD}\ \ service\ \ rid\ \ \rho_{\mathsf{message\_info}}\ \ \rho_{\mathsf{local\_env}}) :: s, headers, \sigma')\end{array}}$$

The call to forward eventually produces a call to forward_response. The semantics of this is shown above, in the treatment of `forward`.

List Creation

List creation is associated with one continuation : `LIST`.

$$\tau, \rho, s \models_{\text{e}} <\ \ exp\ \ > \Rightarrow \tau, \rho, \texttt{LIST} :: s \models_{\text{e}} exp$$

$$\tau, \rho \models_{\text{ec}} \texttt{LIST} :: s, v \Rightarrow \tau, \rho \models_{\text{ec}} s, \langle v \rangle$$

Equality

Equality is associated with two continuations : `EQ1` exp_2 and `EQ2` v_1. When one argument is a response and the other is a code constant, the equality use the response code hierarchy. A response is equal to a code constant if the code response is strictly equal or is in the sub-tree of the code constant hierarchy.

$$\tau, \rho, s \models_{\text{e}} exp_1\ \ ==\ \ exp_2 \Rightarrow \tau, \rho, (\texttt{EQ1}\ \ exp_2) :: s \models_{\text{e}} exp_1$$

$$\tau, \rho \models_{\text{ec}} (\texttt{EQ1}\ \ exp_2) :: s, v_1 \Rightarrow \tau, \rho, (\texttt{EQ2}\ \ v_1) :: s \models_{\text{e}} exp_2$$

$$\tau, \rho \models_{\text{ec}} (\texttt{EQ2}\ \ v_1) :: s, v_2 \Rightarrow \tau, \rho \models_{\text{ec}} s, v_1 == v_2$$

Identifier

$$\tau, \rho, s \models_{e} x \Rightarrow \tau, \rho \models_{ec} s, \mathsf{lookup}(x, \rho)$$

$\mathsf{lookup}(x, \langle envs, \langle m, headers\rangle, local\_\ env\rangle) = \mathsf{lookup}_l(x, envs, \langle m, headers\rangle, local\_\ env)$, which is defined as follows :

$$\frac{x \in \mathsf{dom}(local\_\ env)}{\mathsf{lookup}_l(x, envs, message\_\ info, local\_\ env) = local\_\ env(x)}$$

$$\frac{x \notin \mathsf{dom}(local\_\ env)}{\mathsf{lookup}_l(x, envs, message\_\ info, local\_\ env) = \mathsf{lookup}_h(x, envs, message\_\ info)}$$

$$\frac{x \in \mathsf{dom}(headers)}{\mathsf{lookup}_h(x, envs, \langle rq, headers\rangle) = headers(x)}$$

$$\frac{x \notin \mathsf{dom}(headers)}{\mathsf{lookup}_h(x, envs, \langle rq, headers\rangle) = \mathsf{lookup}_e(x, envs)}$$

$$\frac{x \in \mathsf{dom}(env)}{\mathsf{lookup}_e(x, envs +\!\!+ \langle env\rangle) = env(x)} \qquad \frac{x \notin \mathsf{dom}(env)}{\mathsf{lookup}_e(x, envs +\!\!+ \langle env\rangle) = \mathsf{lookup}_e(x, envs)}$$

Constant

$$\tau, \rho, s \models_{e} c \Rightarrow \tau, \rho \models_{ec} s, c$$

RequestURI

$$\frac{\rho_{\mathsf{message\_info}} = \langle rq, headers\rangle}{\tau, \rho, s \models_{e} \texttt{requestURI} \Rightarrow \tau, \rho \models_{ec} s, rq}$$

This

$$\frac{\rho_{\mathsf{message\_info}} = \langle rq, headers\rangle}{\tau, \rho, s \models_{e} \texttt{this} \Rightarrow \tau, \rho \models_{ec} s, \langle rq, headers\rangle}$$

With

We assume there is only one binding, as multiple bindings can be implemented by nested `withs`. There are two continuations : `WITH1` *field* exp_2 and `WITH2` *field resp*.

$$\tau, \rho, s \models_{\mathrm{e}} exp_1 \ \texttt{with} \ field \ \texttt{=} \ exp_2 \Rightarrow \tau, \rho, (\texttt{WITH1} \ field \ exp_2) :: s \models_{\mathrm{e}} exp_1$$

$$\tau, \rho \models_{\mathrm{ec}} (\texttt{WITH1} \ field \ exp_2) :: s, v \Rightarrow \tau, \rho, (\texttt{WITH2} \ field \ v) :: s \models_{\mathrm{e}} exp_2$$

$$\tau, \rho \models_{\mathrm{ec}} (\texttt{WITH2} \ field \ code\,(resp\_ \ headers)\,) :: s, v \Rightarrow \tau, \rho \models_{\mathrm{ec}} s, code(resp\_ \ headers[field \mapsto v])$$

B.7.7 Semantics of Declarations

The semantics of declarations is defined as below. Each expression is evaluated with respect to the empty continuation, because we know that there is no `forward` in a declaration, even the declaration of a local variable. This is always invoked with $\emptyset$ in the *local_ env* position of ρ.

$$\langle envs, message\_ \ info, local\_ \ env\rangle \models_{\mathrm{d}} \langle\rangle \Rightarrow local\_ \ env$$

$$\frac{\begin{array}{c}\bot, \rho, \langle\rangle \models_{\mathrm{e}} exp_1 \Rightarrow^* \_, \_, \_ \models_{\mathrm{ec}} \langle\rangle, v_1 \\ \rho = \langle envs, message\_ \ info, local\_ \ env\rangle \\ \langle envs, message\_ \ info, local\_ \ env[x_1 \mapsto v_1]\rangle \models_{\mathrm{d}} \langle\langle x_2, exp_2\rangle, \ldots\rangle \Rightarrow env\end{array}}{\rho \models_{\mathrm{d}} \langle\langle x_1, exp_1\rangle, \langle x_2, exp_2\rangle, \ldots\rangle \Rightarrow env}$$

Annexe C

Service de file d'attente en SPL

```
1  // Call Queuing (QUEUE)
2
3  service queue {
4    processing {
5
6      extern string sip_invite ( uri to );
7      extern void sip_bye ( string call_id );
8
9      extern void rtp_play ( string call_id, string file );
10     extern void rtp_bridge ( string call_id_from, string call_id_to );
11
12     extern int hashtbl_create ( int size );
13     extern void hashtbl_add ( int assoc, string key, string value );
14     extern string hashtbl_find ( int assoc, string key );
15     extern void hashtbl_delete ( int assoc, string key );
16     extern void hashtbl_clean ( int assoc );
17     extern int hashtbl_length ( int assoc );
18
19
20     FIFO string<4> remove_element ( string call_id, FIFO string<4> list ) {
21       FIFO string<4> temp_waiting_list;
22       foreach ( waiting_session in list ) {
23         if ( waiting_session != call_id ) {
24           push temp_waiting_list waiting_session;
25         }
26       }
27       return temp_waiting_list;
28     }
29
30     void play(string call_id, int index) {
31       select(index) {
32       case 0: rtp_play(call_id, "attendee.wav"); return;
33       case 1: rtp_play(call_id, "attendee1.wav"); return;
34       case 2: rtp_play(call_id, "attendee2.wav"); return;
35       case 3: rtp_play(call_id, "attendee3.wav"); return;
36       case 4: rtp_play(call_id, "attendee4.wav"); return;
37       default: rtp_play(call_id, "attendee5+.wav"); return;
38       }
39     }
40
41     int callers;
42     int callees;
43     int callees_temp;
44
45     uri callee = 'sip:secretaire@enseirb.fr';
46
47     void deploy () {
48       callers = hashtbl_create ( 5 );
```

```
49        callees = hashtbl_create ( 1 );
50        callees_temp = hashtbl_create ( 5 );
51      }
52
53      void undeploy () {
54        hashtbl_clean ( callers );
55        hashtbl_clean ( callees );
56        hashtbl_clean ( callees_temp );
57      }
58
59      registration {
60
61        int presence = 0;  // -1: undef, 0: available, 1: busy
62
63        FIFO string<4> waiting_list;
64        int currentSize = 4;
65
66        dialog {
67
68          response incoming INVITE () {
69            if ( presence != -1 ) {
70              if ( currentSize != 0 ) {
71                response resp = forward;
72                if ( resp == /SUCCESS ) {
73                  currentSize = currentSize - 1;
74                  return resp;
75                } else {
76                  return /ERROR/SERVER/SERVICE_UNAVAILABLE;
77                }
78              } else {
79                return /ERROR/CLIENT/TEMPORARILY_UNAVAILABLE
80                  with { reason = "Too many callers in hold, please retry later..." };
81              }
82            } else {
83              return /ERROR/SERVER/SERVICE_UNAVAILABLE;
84            }
85          }
86
87          void incoming ACK () {
88            string session_id = "";
89            if ( waiting_list == FIFO <> ) {
90              push waiting_list CALL_ID;
91              forward;
92              if ( presence == 0 ) {
93                session_id = sip_invite ( callee );
94                hashtbl_add ( callees_temp, session_id, CALL_ID );
95              } else {
96                play ( CALL_ID, hashtbl_length(callers));
97                hashtbl_add ( callers, CALL_ID, session_id );
98              }
99            } else {
100             push waiting_list CALL_ID;
101             forward;
102             play ( CALL_ID, hashtbl_length(callers));
103             hashtbl_add ( callers, CALL_ID, session_id );
104           }
105         }
106
107         response outgoing INVITE () {
108           response res = forward;
109           if ( res == /ERROR ) {
110             string call_id_associated = hashtbl_find ( callees_temp, CALL_ID );
111             hashtbl_delete ( callees_temp, CALL_ID );
112             play ( call_id_associated, hashtbl_length(callers));
113             hashtbl_add ( callers, call_id_associated, "" );
114           }
```

```
115           return res;
116         }
117
118         void outgoing ACK () {
119           forward;
120           presence = 1;
121           string call_id_associated = hashtbl_find ( callees_temp, CALL_ID );
122           waiting_list = remove_element ( call_id_associated, waiting_list );
123           currentSize = currentSize + 1;
124           hashtbl_add ( callers, call_id_associated, CALL_ID );
125           hashtbl_add ( callees, CALL_ID, call_id_associated );
126           hashtbl_delete ( callees_temp, CALL_ID );
127           rtp_bridge ( call_id_associated, CALL_ID);
128
129           int i = 0;
130           foreach(c in waiting_list) {
131             play(c, i);
132             i = i+1;
133           }
134         }
135
136         response incoming BYE () {
137           if ( FROM != callee ) {
138             string call_id_associated = hashtbl_find ( callers, CALL_ID );
139             if ( call_id_associated != "" ) {
140               sip_bye ( call_id_associated );
141               hashtbl_delete ( callers, CALL_ID );
142               return forward;
143             } else {
144               waiting_list = remove_element ( CALL_ID, waiting_list );
145               currentSize = currentSize + 1;
146               hashtbl_delete ( callers, CALL_ID );
147               return forward;
148             }
149           } else {
150             string call_id_associated = hashtbl_find ( callees, CALL_ID );
151             sip_bye ( call_id_associated );
152             hashtbl_delete ( callees, CALL_ID );
153             response res = forward;
154             presence = 0;
155             if ( waiting_list != FIFO <> ) {
156               string next_session = top waiting_list;
157               hashtbl_delete ( callers, next_session );
158               string session_id = sip_invite ( callee );
159               hashtbl_add ( callees_temp, session_id, next_session );
160             }
161             return res;
162           }
163         }
164
165         response outgoing BYE () {
166           if ( TO != callee ) {
167             hashtbl_delete ( callers, CALL_ID);
168             return forward;
169           } else {
170             response res = forward;
171             hashtbl_delete ( callees, CALL_ID );
172             presence = 0;
173             if ( waiting_list != FIFO <> ) {
174               string next_session = top waiting_list;
175               hashtbl_delete ( callers, next_session );
176               string session_id = sip_invite ( callee );
177               hashtbl_add ( callees_temp, session_id, next_session );
178             }
179             return res;
180 } } } } } }
```

Annexe D

Pantaxou Language

D.1 Domain Syntax

A description of an application domain has the following form.

```
struct structName (extends parentStruct)? {
  (type id;)*
}
enum enumName = {id1, ..., idn}
command commandName {
  (type id(type1 id1, ..., typen idn) ;)+
}
service serviceName (extends parentService)? {
  (type id;)*
  ((requires|provides) command<commandName>(from|to) serviceName;)*
  ((requires|provides) event<datatypeName>(from|to) serviceName;)*
  ((requires|provides) session<(?|!|=)datatypeName>(from|to) serviceName;)*
}
```

The datatype taxonomy stands for a forest, while the service taxonomy is a tree whose the root must be named *Service*. A command consists of a group of methods.

The grammar is defined as follows :

```
domain           ::= datatype* commandInterface* service
datatype         ::= import identifier
                   | struct identifier (extends identifier)? {property* }
                   | enum identifier = {identifier (, identifier)+ }
commandInterface ::= command identifier {method* }
service          ::= service identifier (extends identifier)? {property* functionality* }
method           ::= type identifier ( (type identifier (, type identifier)*)? ) ;
property         ::= type identifier ;
functionality    ::= requires interactionMode from identifier ;
                   | provides interactionMode to identifier ;
                   | bind<functionality, functionality> ;
```

```
interactionMode ::= command<identifier>
                  | event<identifier>
                  | session<(?|!|=) identifier>
type            ::= identifier
                  | primitiveType
primitiveType   ::= bool | int | string | uri | void
```

D.2 Domain Static Semantics

The static semantics uses the following type system and environments :

Type System

The + operator is used to build disjoint unions.
PrimitiveType = Bool + Int + String + URI
PrimitiveTypeWithVoid = PrimitiveType + Void
Direction = {Requires, Provides}
Command = Identifier
Event = Identifier
Session = Identifier
InteractionMode = Command + Event + Session
Bind = (Direction × Session × Identifier) × (Direction × Session × Identifier)
Datatype = Identifier
Type = PrimitiveType + Datatype
TypeWithVoid = PrimitiveTypeWithVoid + Datatype

Environments and Sets

For an environment e, the function dom(e) returns the domain of e.

e_m is a method environment.
e_m : Identifier → (TypeWithVoid × (Type list))

e_c is a command environment.
e_c : Identifier → e_m

e_p is a property environment.
e_p : Identifier → Type
s_f is a functionality set.
s_f consists of triplet (Direction × InteractionMode × Identifier) and Bind

e_d is a datatype environment.
e_d : Identifier → ((Identifier ∪ ∅) × e_p)

e_s is a service environment.
e_s : Identifier → ((Identifier ∪ ∅) × e_p × s_f)

D.2.1 Static Semantics

$$\frac{}{e_d \vdash \texttt{bool} \ : \ \mathsf{Bool}} \qquad \frac{}{e_d \vdash \texttt{int} \ : \ \mathsf{Int}} \qquad \frac{}{e_d \vdash \texttt{string} \ : \ \mathsf{String}}$$

$$\frac{}{e_d \vdash \texttt{uri} \ : \ \mathsf{URI}} \qquad \frac{}{e_d \vdash \texttt{void} \ : \ \mathsf{Void}} \qquad \frac{id \in \mathrm{dom}(e_d)}{e_d \vdash id \ : \ \mathsf{Datatype}(id)}$$

$$\frac{id \in \mathrm{dom}(e_c)}{e_d,\ e_c \vdash \texttt{command}\langle id \rangle \ : \ \mathsf{Command}(id)}$$

$$\frac{id \in \mathrm{dom}(e_d)}{e_d,\ e_c \vdash \texttt{event}\langle id \rangle \ : \ \mathsf{Event}(id)}$$

$$\frac{id \in \mathrm{dom}(e_d)}{e_d,\ e_c \vdash \texttt{session}\langle id \rangle \ : \ \mathsf{Session}(id)}$$

$$\frac{e_d,\ e_c \vdash interactionmode \ : \ im}{e_d,\ e_c,\ s_f \vdash \texttt{requires}\ interactionmode\ \texttt{from}\ id \ : \ s_f \cup \{(\mathsf{Requires},\ im,\ id)\}}$$

$$\frac{e_d,\ e_c \vdash interactionmode \ : \ im}{e_d,\ e_c,\ s_f \vdash \texttt{provides}\ interactionmode\ \texttt{to}\ id \ : \ s_f \cup \{(\mathsf{Provides},\ im,\ id)\}}$$

$$\frac{\begin{array}{c}\emptyset \vdash functionality_1 \ : \ s_{f_1} \qquad \#(s_{f_1}) = 1 \qquad (d_1,\ \mathsf{Session}(t_1),\ s_1) \in s_{f_1} \\ \emptyset \vdash functionality_2 \ : \ s_{f_2} \qquad \#(s_{f_2}) = 1 \qquad (d_2,\ \mathsf{Session}(t_2),\ s_2) \in s_{f_2} \\ t_1 \equiv t_2 \ (t_1 = t_2 \text{ or } t_1 \text{ is a subtype of } t_2 \text{ or } t_2 \text{ is a subtype of } t_1) \\ s'_f = s_f \cup \mathsf{Bind}((d_1,\ \mathsf{Session}(t_1),\ s_1), (d_2,\ \mathsf{Session}(t_2),\ s_2))\end{array}}{e_d,\ e_c,\ s_f \vdash \texttt{bind}\langle functionality_1,\ functionality_2 \rangle \ : \ s'_f}$$

$$\frac{e_d,\ e_c,\ s_f \vdash functionality_1 \ : \ s_{f_1} \cdots e_d,\ e_c,\ s_{f_{n-1}} \vdash functionality_n \ : \ s_{f_n}}{e_d,\ e_c,\ s_f \vdash functionality_1 \cdots functionality_n \ : \ s_{f_n}}$$

$$\frac{e_d \vdash type \ : \ t \qquad t \neq \mathsf{Void}}{e_d,\ e_p \vdash type\ id; \ : \ e_p[id \mapsto t]}$$

$$\frac{e_d,\ e_p \vdash property_1 \ : \ e_{p_1} \cdots e_d,\ e_{p_{n-1}} \vdash property_n \ : \ e_{p_n}}{e_d,\ e_p \vdash property_1 \cdots property_n \ : \ e_{p_n}}$$

$$\frac{id_2 \in \mathrm{dom}(e_s) \qquad e_d,\ \varnothing \vdash properties \ : \ e_p \qquad e_d,\ e_c,\ \emptyset \vdash functionalities \ : \ s_f}{e_d,\ e_c,\ e_s \vdash \texttt{service}\ id_1\ \texttt{extends}\ id_2\ \{\ properties\ functionalities\ \} \ : \ e_s[id_1 \mapsto (id_2,\ e_p,\ s_f)]}$$

$$\frac{e_d,\ \varnothing \vdash properties \ : \ e_p \qquad e_d,\ e_c,\ \emptyset \vdash functionalities \ : \ s_f}{e_d,\ e_c,\ e_s \vdash \texttt{service}\ id\ \{\ properties\ functionalities\ \} \ : \ e_s[id \mapsto (\varnothing,\ e_p,\ s_f)]}$$

$$\frac{e_d,\ e_c,\ e_s \vdash service_1 \ : \ e_{s_1} \cdots e_d,\ e_c,\ e_{s_{n-1}} \vdash service_n \ : \ e_{s_n}}{e_d,\ e_c,\ e_s \vdash service_1 \cdots service_n \ : \ e_{s_n}}$$

$$\frac{e_d \vdash type \ : \ t \qquad e_d \vdash type_1 \ : \ t_1 \ \ ... \ \ e_d \vdash type_n \ : \ t_n \qquad \forall i \in [1..n] \cdot t_i \neq \mathsf{Void}}{e_d, \ e_m \vdash type \ id(type_1 \ id_1, ..., type_n \ id_n); \ : \ e_m[id \mapsto (t, \ \langle t_1, ..., t_n \rangle)]}$$

$$\frac{e_d, \ e_m \vdash method_1 \ : \ e_{m_1} \cdots e_d, \ e_{m_{n-1}} \vdash method_n \ : \ e_{m_n}}{e_d, \ e_m \vdash method_1 \cdots method_n \ : \ e_{m_n}}$$

$$\frac{e_d, \ \varnothing \vdash methods \ : \ e_m}{e_d, \ e_c \vdash \texttt{command} \ id \ \{ \ methods \ \} \ : \ e_c[id \mapsto e_m]}$$

$$\frac{e_d, \ e_c \vdash command_1 \ : \ e_{c_1} \cdots e_d, \ e_{c_{n-1}} \vdash command_n \ : \ e_{c_n}}{e_d, \ e_c \vdash command_1 \cdots command_n \ : \ e_{c_n}}$$

$$\frac{id_2 \in \mathrm{dom}(e_d) \qquad e_d, \ \varnothing \vdash properties \ : \ e_p}{e_d \vdash \texttt{struct} \ id_1 \ \texttt{extends} \ id_2 \ \{ \ properties \ \} \ : \ e_d[id_1 \mapsto (id_2, \ e_p)]}$$

$$\frac{e_d, \ \varnothing \vdash properties \ : \ e_p}{e_d \vdash \texttt{struct} \ id \ \{ \ properties \ \} \ : \ e_d[id \mapsto (\varnothing, \ e_p)]}$$

$$\frac{e_d \vdash datatype_1 \ : \ e_{d_1} \cdots e_{d_{n-1}} \vdash datatype_n \ : \ e_{d_n}}{e_d \vdash datatype_1 \cdots datatype_n \ : \ e_{d_n}}$$

$$\frac{\varnothing \vdash datatypes \ : \ e_d \qquad e_d, \ \varnothing \vdash commands \ : \ e_c \qquad e_d, \ e_c, \ \varnothing \vdash services \ : \ e_s}{\vdash datatypes \ commands \ services \ : \ (e_d, \ e_c, \ e_s)}$$

D.3 Service Syntax

A Πανταχου program has the following form.

```
'uri' instantiates servicePath {
  service<servicePath> {
    (property_name = exp;)*
  } service_name;
  (event|session)<userTypePath> (from service_name)? {
    (property_name = exp;)*
  } interaction_name;
  (type id = exp;)*
  (command type cmd_name(param1 ... paramn) {stmt*})*
  onReceive behavior_name(interaction_name id) {stmt*};
  initial {decl* stmt*}
  final {decl* stmt*}
  onError {decl* stmt*}
}
```

The grammar is defined as follows :

```
program      ::= import* 'uri_name' instantiates servicePath {body}
import       ::= import ontology identifier;
               | import command identifier;
               | import service servicePath;
               | import datatype userTypePath;
body         ::= decl* initial {stmt*} final {stmt*} onError {stmt*}
decl         ::= var type identifier = exp;
               | service<servicePath> {property*} identifier;
               | mode<userTypePath> (source)? {property*} identifier;
               | bridge<type> {decl*} identifier;
               | command type identifier((param (, param)*)?) {stmt*}
               | function type identifier((param (, param)*)?) {stmt*}
               | onReceive identifier (param) {stmt*}
type         ::= simpleType([integer?])?
               | activeSession<userTypePath>
               | identifier([integer?])?
               | userTypePath
simpleType   ::= bool | int | string | uri | void
const        ::= true | false | integer | "string" | 'uri'
servicePath  ::= Service (.identifier)*
userTypePath ::= identifier (.identifier)*
property     ::= (identifier.)?identifier = exp;
               | identifier = userTypePath{property*};
source       ::= from identifier[integer | *]
mode         ::= event | session
param        ::= type identifier
```

```
stmt  ::= decl
        | exp ;
        | place = exp ;
        | if ( exp ) {stmt*}
        | if ( exp ) {stmt*} else {stmt*}
        | return exp? ;
        | invite(exp, exp) {accepted(param) {stmt*} rejected() {stmt*}}
        | publish(exp) ;
        | adopt(exp) ;
        | reject(exp) ;
        | disconnect(exp) ;
exp   ::= const | new type([integer | *])?
        | exp.bind(exp, exp) | accept (exp)
        | exp.identifier ( (exp (, exp)*)? )
        | identifier ( (exp (, exp)*)? )
        | identifier | exp.identifier
place ::= identifier | exp.identifier
```

D.4 Service Static Semantics

Type System

In order to evaluate a Πανταχου program, the domain Γ to which it applies must be given. Γ is a triplet (Σ, Ξ, Δ) which respectively defines for the Γ domain its services Σ of type **ServiceType**, its commands Ξ of type **Command** and its datatypes Δ of type **UserType**. **ServiceType** is a hierarchical set of types defined by an ontology. **Command** is a set of commands which defines some method signatures. **UserType** is a hierarchical set of types with some properties attached to each node.

SimpleType : Void + Bool + Int + String + URI $\rightarrow$ *ty*
DataType : SimpleType + **UserType** $\rightarrow$ *ty*
UserType : Identifier list $\times$ (DataType $\rightarrow$ Identifier) $\rightarrow$ *ty*
Command : Identifier $\times$ (Identifier $\times$ DataType list $\rightarrow$ DataType) $\rightarrow$ *ty*
RequiredEventType, ProvidedEventType : Identifier list $\rightarrow$ *ty*
ProvidedSessionType, RequiredSessionType : Identifier list $\rightarrow$ *ty*
EventInstType, SessionInstType : Identifier list $\rightarrow$ *ty*
EventReceptionType, SessionReceptionType : Identifier $\rightarrow$ *ty*
BridgeType, BridgeInstType, ActiveSessionType : Identifier list $\rightarrow$ *ty*
ModeType : RequiredEventType + ProvidedEventType
+ ProvidedSessionType + RequiredSessionType
+ ProvidedCommandType + RequiredCommandType $\rightarrow$ *ty*
ProvidedCommandType, RequiredCommandType : Identifier $\rightarrow$ *ty*
ListType : DataType $\rightarrow$ *ty*
ServiceType : Identifier list $\times$ ModeType list $\times$ (Identifier $\rightarrow$ DataType)list $\rightarrow$ *ty*
ServiceInstType : $(\mathbb{N} \cup \top) \times$ **ServiceType** $\rightarrow$ *ty*

$$
\begin{array}{rcl}
ty & : & \mathsf{SimpleType} + \mathsf{DataType} + \mathsf{ListType} + \mathsf{RequiredEventType} + \mathsf{ProvidedEventType} \\
& + & \mathsf{ProvidedSessionType} + \mathsf{RequiredSessionType} + \mathsf{ActiveSessionType} + \mathsf{ModeType} \\
& + & \mathbf{Command} + \mathsf{ProvidedCommandType} + \mathsf{RequiredCommandType} + \mathsf{BridgeType} \\
& + & \mathsf{EventReceptionType} + \mathsf{SessionReceptionType} + \mathbf{ServiceType} + \mathbf{UserType} \\
& + & \mathsf{ServiceInstType} + \mathsf{EventInstType} + \mathsf{SessionInstType} + \mathsf{BridgeInstType}
\end{array}
$$

Domain Description

Reminder : domain $\Gamma = (\Sigma, \Xi, \Delta)$

Σ is a environment which maps service label to its definition.
Σ : Identifier list $\rightarrow$ **ServiceType**

Ξ is a environment which maps command name to the list of methods it contains. As method signatures are unique and belongs to a unique command name, Ξ^{-1} is also defined as following.
Ξ : Identifier $\rightarrow$ **Command** list
Ξ^{-1} : Identifier $\times$ DataType list $\rightarrow$ **Command**

Δ is a environment which maps user data type label to its definition.
Δ : Identifier list $\rightarrow$ **UserType**

Relations

$\sqsubseteq$ is the subclass relation. It can be applied on **ServiceType** and **UserType**.
$\sqsubseteq$: **ServiceType** $\times$ **ServiceType** $\sqsubseteq$: **UserType** $\times$ **UserType**
Then, the equivalence relation, $\equiv$, can be defined as follows : $\mathsf{X} \equiv \mathsf{Y} \Leftrightarrow \mathsf{X} \sqsubseteq \mathsf{Y} \vee \mathsf{Y} \sqsubseteq \mathsf{X}$

$\oplus : ty \times ty \rightarrow ty$ $\qquad \dfrac{t_1 = \mathsf{Void} \vee t_2 = \mathsf{Void}}{t_1 \oplus t_2 = \mathsf{Void}} \qquad \dfrac{t_1 = t_2 = t' \neq \mathsf{Void}}{t_1 \oplus t_2 = t'}$

Operator

The operator $+\!\!+$ is used to concatenate lists.

D.4.1 Static Semantics

The type system is defined in terms of a domain Γ, an environment μ which contains the ModeType set provided by the service and an environment τ of type *identifier* $\rightarrow$ *ty*. The judgments for statements, expressions, declarations and programs have the following form, where $t \in ty$:

- $\Gamma,\ \mu,\ \tau,\ \tau_{\mathsf{prop}} \vdash_{\mathrm{p}} properties$
- $\Gamma,\ \mu,\ \tau,\ t \vdash_{\mathrm{s}} stmt : t'$
- $\Gamma,\ \mu,\ \tau \vdash_{\mathrm{d}} decl : \tau'$
- $\Gamma,\ \mu,\ \tau \vdash_{\mathrm{m}} method$
- $\Gamma,\ \mu,\ \tau \vdash_{\mathrm{t}} type : t$
- $\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp : t$
- $\Gamma \vdash_{\mathrm{pg}} program$

The judgments for service, user-type and modes properties are defined as follows :

$$\frac{\Gamma,\ \mu,\ \tau \cup \tau_{\mathsf{prop}} \vdash_{\mathrm{e}} exp : t \qquad t = \tau_{\mathsf{prop}}(p)}{\Gamma,\ \mu,\ \tau,\ \tau_{\mathsf{prop}} \vdash_{\mathrm{p}} p\ =\ exp}$$

$$\frac{\Gamma \vdash_{\mathrm{upa}} \mathit{UserTypePath} : t \qquad t = \tau_{\mathsf{prop}}(p) \qquad t = \mathbf{UserType}(\_, \tau'_{\mathsf{prop}}) \qquad \Gamma,\ \mu,\ \tau,\ \tau'_{\mathsf{prop}} \vdash_{\mathrm{p}} \mathit{properties}}{\Gamma,\ \mu,\ \tau,\ \tau_{\mathsf{prop}} \vdash_{\mathrm{p}} p\ =\ \mathit{UserTypePath}\ \ \{\mathit{properties}\ \ \}}$$

$$\frac{\Gamma,\ \mu,\ \tau,\ \tau_{\mathsf{prop}} \vdash_{\mathrm{p}} pe_1 \qquad \cdots \qquad \Gamma,\ \mu,\ \tau,\ \tau_{\mathsf{prop}} \vdash_{\mathrm{p}} pe_p}{\Gamma,\ \mu,\ \tau,\ \tau_{\mathsf{prop}} \vdash_{\mathrm{p}} pe_1\ \texttt{;}\ \ \ldots\ \texttt{;}\ \ pe_p\ \texttt{;}}$$

The judgments for service types are defined as follows :

$$\frac{\Sigma(\langle \texttt{Service} \rangle) = t \qquad t = \mathbf{ServiceType}(\langle \texttt{Service} \rangle, \_, \_)}{(\Sigma, \Xi, \Delta) \vdash_{\mathrm{spa}} \texttt{Service} : t}$$

$$\frac{(\Sigma, \Xi, \Delta) \vdash_{\mathrm{spa}} \mathit{pServiceTypePath} : \mathbf{ServiceType}(\mathit{serviceTypeLabel}, \_, \_) \qquad \Sigma(\mathit{serviceTypeLabel} +\!\!+ \langle \mathit{identifier} \rangle) = t \qquad t = \mathbf{ServiceType}(\mathit{serviceTypeLabel} +\!\!+ \langle \mathit{identifier} \rangle, \_, \_)}{(\Sigma, \Xi, \Delta) \vdash_{\mathrm{spa}} \mathit{pServiceTypePath}\ \ \texttt{.}\ \ \mathit{identifier} : t}$$

The judgments for user data types are defined as follows :

$$\frac{\Delta(\langle \mathit{identifier} \rangle) = t \qquad t = \mathbf{UserType}(\langle \mathit{identifier} \rangle, \_)}{(\Sigma, \Xi, \Delta) \vdash_{\mathrm{upa}} \mathit{identifier} : t}$$

$$\frac{(\Sigma, \Xi, \Delta) \vdash_{\mathrm{upa}} \mathit{pUserTypePath} : \mathbf{UserType}(\mathit{userTypeLabel}, \_) \qquad \Delta(\mathit{userTypeLabel} +\!\!+ \langle \mathit{identifier} \rangle) = t \qquad t = \mathbf{UserType}(\mathit{userTypeLabel} +\!\!+ \langle \mathit{identifier} \rangle, \_)}{(\Sigma, \Xi, \Delta) \vdash_{\mathrm{upa}} \mathit{pUserTypePath}\ \ \texttt{.}\ \ \mathit{identifier} : t}$$

The judgments for simple types are defined as follows :

$$\frac{}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{t}} \texttt{void} : \mathsf{Void}} \qquad \frac{}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{t}} \texttt{bool} : \mathsf{Bool}}$$

$$\frac{}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{t}} \texttt{int} : \mathsf{Int}} \qquad \frac{}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{t}} \texttt{string} : \mathsf{String}} \qquad \frac{}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{t}} \texttt{uri} : \mathsf{URI}}$$

$$\frac{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{t}} \mathit{userTypeLabel} : t}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{t}} \texttt{activeSession<}\mathit{userTypeLabel}\texttt{>} : \mathsf{ActiveSessionType}(t)}$$

The judgments for compounds are defined as follows :

$$\frac{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{d}} \mathit{decl}_1 \ldots \mathit{decl}_m : \tau' \qquad \Gamma,\ \mu,\ \tau',\ t \vdash_{\mathrm{s}} \mathit{stmt}_1 : t_1 \qquad \cdots \qquad \Gamma,\ \mu,\ \tau',\ t \vdash_{\mathrm{s}} \mathit{stmt}_n : t_n \qquad \forall j < i, t_j = \mathsf{Void} \qquad i < n \wedge t_i \neq \mathsf{Void} \bigvee i = n}{\Gamma,\ \mu,\ \tau,\ t \vdash_{\mathrm{s}} \{\, \mathit{decl}_1\ \ \ldots\ \ \mathit{decl}_m\ \ \mathit{stmt}_1\ \ \mathit{stmt}_n \,\} : t_i}$$

$$\frac{}{\Gamma,\ \mu,\ \tau,\ t \vdash_{\mathrm{s}} \{\,\} : \mathsf{Void}}$$

The judgments for statements are defined as follows :

$$\frac{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} place : t \qquad \Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp : t' \qquad t = t'}{\Gamma,\ \mu,\ \tau,\ \_ \vdash_{\mathrm{s}} place\ =\ exp\texttt{ ; } : \mathsf{Void}}$$

$$\frac{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp : \mathsf{Void}}{\Gamma,\ \mu,\ \tau,\ t \vdash_{\mathrm{s}} exp\texttt{ ; } : \mathsf{Void}} \qquad \frac{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp : \mathsf{ListType}(\mathsf{Void})}{\Gamma,\ \mu,\ \tau,\ t \vdash_{\mathrm{s}} exp\texttt{ ; } : \mathsf{Void}}$$

$$\frac{t = \mathsf{Void}}{\Gamma,\ \mu,\ \tau,\ t \vdash_{\mathrm{s}} \texttt{return ; } : t} \qquad \frac{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp : t' \qquad t = t'}{\Gamma,\ \mu,\ \tau,\ t \vdash_{\mathrm{s}} \texttt{return }\ exp\texttt{ ; } : t}$$

$$\frac{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp : \mathsf{Bool} \qquad \Gamma,\ \mu,\ \tau,\ t \vdash_{\mathrm{s}} cmpd_1 : t_1 \qquad \Gamma,\ \mu,\ \tau,\ t \vdash_{\mathrm{s}} cmpd_2 : t_2}{\Gamma,\ \mu,\ \tau,\ t \vdash_{\mathrm{s}} \texttt{if }\ (exp)\ \ cmpd_1\ \ \texttt{else }\ cmpd_2 : t_1 \oplus t_2}$$

$$\frac{\begin{array}{c}\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp_1 : \mathbf{ServiceType}(\_, modes, \_) \\ \Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp_2 : \mathsf{RequiredSessionType}(userTypeLabel) \\ \exists userTypeLabel', \Delta(userTypeLabel') \sqsubseteq \Delta(userTypeLabel) \\ \wedge \mathsf{ProvidedSessionType}(userTypeLabel') \in modes \\ type = \mathsf{ActiveSessionType}(userTypeLabel) \\ \Gamma,\ \mu,\ \tau[id \mapsto type],\ t \vdash_{\mathrm{s}} cmpd_{acc} : t_{acc} \qquad \Gamma,\ \mu,\ \tau,\ t \vdash_{\mathrm{s}} cmpd_{rej} : t_{rej} \\ t_{acc} = t_{rej} = t'\end{array}}{\begin{array}{l}\Gamma,\ \mu,\ \tau,\ t \vdash_{\mathrm{s}} \texttt{invite(}exp_1\texttt{, }\ exp_2\texttt{)\{accepted(}type\ id\texttt{)}\ \ cmpd_{acc} \\ \qquad\qquad \texttt{rejected()}\ \ cmpd_{rej}\ \texttt{\}} : t'\end{array}}$$

Note : invite should handle both cases, accept and reject. The reason is the reject statement should not be viewed as an error by the remote service.

$$\frac{\begin{array}{c}\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp_1 : \mathbf{ServiceType}(\_, modes, \_) \\ \Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp_2 : \mathsf{SessionInstType}(userTypeLabel) \\ \exists userTypeLabel', \Delta(userTypeLabel') \sqsubseteq \Delta(userTypeLabel) \\ \wedge\ \mathsf{ProvidedSessionType}(userTypeLabel') \in modes \\ type = \mathsf{ActiveSessionType}(userTypeLabel) \\ \Gamma,\ \mu,\ \tau[id \mapsto type],\ t \vdash_{\mathrm{s}} cmpd_{acc} : t_{acc} \qquad \Gamma,\ \mu,\ \tau,\ t \vdash_{\mathrm{s}} cmpd_{rej} : t_{rej} \\ t_{acc} = t_{rej} = t'\end{array}}{\begin{array}{l}\Gamma,\ \mu,\ \tau,\ t \vdash_{\mathrm{s}} \texttt{invite(}exp_1\texttt{, }\ exp_2\texttt{)\{accepted(}type\ id\texttt{)}\ \ cmpd_{acc} \\ \qquad\qquad \texttt{rejected()}\ \ cmpd_{rej}\ \texttt{\}} : t'\end{array}}$$

$$\frac{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp : \mathsf{EventInstType}(etl) \qquad \mathsf{ProvidedEventType}(etl) \in \mu}{\Gamma,\ \mu,\ \tau,\ t \vdash_{\mathrm{s}} \texttt{publish }\ (exp)\texttt{ ; } : \mathsf{Void}}$$

$$\frac{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp : \mathsf{EventReceptionType}(\_) + \mathsf{SessionReceptionType}(\_)}{\Gamma,\ \mu,\ \tau,\ t \vdash_{\mathrm{s}} \texttt{adopt(}exp\texttt{) ; } : \mathsf{Void}}$$

$$\frac{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp : t' \qquad t' = \mathsf{ProvidedSessionType}(\_)}{\Gamma,\ \mu,\ \tau,\ t \vdash_{\mathrm{s}} \texttt{reject(}exp\texttt{) ; } : \mathsf{Void}} \qquad \frac{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp : t' \qquad t' = \mathsf{ActiveSessionType}(\_) + \mathsf{BridgeInstType}(\_)}{\Gamma,\ \mu,\ \tau,\ t \vdash_{\mathrm{s}} \texttt{disconnect(}exp\texttt{) ; } : \mathsf{Void}}$$

The judgments for expressions are defined as follows :

$$\frac{c \in \{\texttt{"}\ldots\texttt{"}, \ldots\}}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} c : \mathsf{String}} \qquad \frac{c \in \{\texttt{'}\ldots\texttt{'}, \ldots\}}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} c : \mathsf{URI}} \qquad \frac{c \in \mathbb{Z}}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} c : \mathsf{Int}} \qquad \frac{c \in \{\texttt{true}, \texttt{false}\}}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} c : \mathsf{Bool}}$$

$$\frac{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp : \mathsf{ProvidedSessionType}(userTypeLabel)}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} \texttt{accept (}exp\texttt{)} : \mathsf{ActiveSessionType}(userTypeLabel)}$$

$$\frac{\begin{array}{c}\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp : \mathsf{BridgeType}(userTypeLabel) \\ \Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp_1 : \mathsf{ActiveSessionType}(userTypeLabel_1) \\ \Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp_2 : \mathsf{ActiveSessionType}(userTypeLabel_2) \\ userTypeLabel_1 \equiv userTypeLabel_2 \\ \Delta(userTypeLabel_1) \sqsubseteq \Delta(userTypeLabel) \wedge \Delta(userTypeLabel_2) \sqsubseteq \Delta(userTypeLabel)\end{array}}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp\texttt{.bind (}exp_1\texttt{, }exp_2\texttt{)} : \mathsf{BridgeInstType}(userTypeLabel)}$$

$$\frac{\begin{array}{c}\tau(identifier) = \mathsf{RequiredSessionType}(t) \\ t' = \mathsf{SessionInstType}(t)\end{array}}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} \texttt{new}\ \ identifier : t'} \qquad \frac{\begin{array}{c}\tau(identifier) = \mathsf{ProvidedEventType}(t) \\ t' = \mathsf{EventInstType}(t)\end{array}}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} \texttt{new}\ \ identifier : t'}$$

$$\frac{\begin{array}{c}\tau(service) = \mathbf{ServiceType}(\_, \_, \_) \\ i \in \mathbb{N}^* \\ \mathsf{ServiceInstType}(i, \tau(service)) = t\end{array}}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} \texttt{new}\ \ service\texttt{[i]} : t} \qquad \frac{\begin{array}{c}\tau(service) = \mathbf{ServiceType}(\_, \_, \_) \\ \mathsf{ServiceInstType}(\top, \tau(service)) = t\end{array}}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} \texttt{new}\ \ service\texttt{[*]} : t}$$

$$\frac{\begin{array}{c}(\Sigma, \Xi, \Delta),\ \mu,\ \tau \vdash_{\mathrm{e}} exp : \mathsf{ServiceInstType}(i, \mathbf{ServiceType}(\_, \mu', \_)) \\ i = 1 \\ (\Sigma, \Xi, \Delta),\ \mu,\ \tau \vdash_{\mathrm{e}} exp_j : t_j,\ \forall j \in \{1 \ldots n\} \\ \Xi^{-1}(cmdName, \langle t_1; \ldots; t_n \rangle) = \mathbf{Command}(cmdLabel, methods) \\ t = methods(cmdName, \langle t_1; \ldots; t_n \rangle) \\ \mathsf{RequiredCommandType}(cmdLabel) \in \mu \qquad \mathsf{ProvidedCommandType}(cmdLabel) \in \mu'\end{array}}{(\Sigma, \Xi, \Delta),\ \mu,\ \tau \vdash_{\mathrm{e}} exp\texttt{.}\,cmdName\texttt{(}exp_1\texttt{, }\ldots\texttt{, }exp_n\texttt{)} : t}$$

$$\frac{\begin{array}{c}(\Sigma, \Xi, \Delta),\ \mu,\ \tau \vdash_{\mathrm{e}} exp : \mathsf{ServiceInstType}(i, \mathbf{ServiceType}(\_, \mu', \_)) \\ i \in (\mathbb{N}^* \cup \{\top\}) \setminus \{1\} \\ (\Sigma, \Xi, \Delta),\ \mu,\ \tau \vdash_{\mathrm{e}} exp_j : t_j,\ \forall j \in \{1 \ldots n\} \\ \Xi^{-1}(cmdName, \langle t_1; \ldots; t_n \rangle) = \mathbf{Command}(cmdLabel, methods) \\ t = methods(cmdName, \langle t_1; \ldots; t_n \rangle) \\ \mathsf{RequiredCommandType}(cmdLabel) \in \mu \qquad \mathsf{ProvidedCommandType}(cmdLabel) \in \mu'\end{array}}{(\Sigma, \Xi, \Delta),\ \mu,\ \tau \vdash_{\mathrm{e}} exp\texttt{.}\,cmdName\texttt{(}exp_1\texttt{, }\ldots\texttt{, }exp_n\texttt{)} : \mathsf{ListType}(t)}$$

The judgments for places are defined as follows :

$$\frac{}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} identifier : \tau(identifier)}$$

$$\frac{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp : \mathsf{ServiceInstType}(i, \mathbf{ServiceType}(\_, \_, fields)) \quad i = 1 \quad fields(field) = t}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp.field : t}$$

$$\frac{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp : \mathsf{ServiceInstType}(i, \mathbf{ServiceType}(\_, \_, fields)) \quad i \in (\mathbb{N}^* \cup \top) \setminus \{1\} \quad fields(field) = t}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp.field : \mathsf{ListType}(t)}$$

The judgments for declarations are defined as follows :

$$\frac{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{d}} decl_1 : \tau_1 \quad \ldots \quad \Gamma,\ \mu, \tau_{n-1} \vdash_{\mathrm{d}} decl_n : \tau_n}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{d}} decl_1\ \ldots\ decl_n : \tau_n} \qquad \frac{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{t}} type : t \quad t \in Datatype(\_) \quad \Gamma,\ \mu,\ \tau \vdash_{\mathrm{e}} exp : t' \quad t = t'}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{d}} \texttt{var}\ \ type\ identifer\ =\ exp\ \texttt{;} : \tau[identifer \mapsto t]}$$

$$\frac{\Gamma,\ \mu,\ \tau,\ \mathsf{Void} \vdash_{\mathrm{s}} cmpd : \mathsf{Void} \quad \Gamma,\ \mu,\ \tau \vdash_{\mathrm{t}} type : \mathbf{UserType}(userTypeLabel)}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{d}} \texttt{bridge<}type\texttt{>}\ \ cmpd\ \ id : \tau[id \mapsto \mathsf{BridgeType}(userTypeLabel)]}$$

$$\frac{\Gamma \vdash_{\mathrm{spa}} servicePath : t \quad t = \mathbf{ServiceType}(\_, \_, \tau_{\mathsf{prop}}) \quad \Gamma,\ \mu,\ \tau, \tau_{\mathsf{prop}} \vdash_{\mathrm{p}} properties}{\Gamma,\ \mu,\ \tau \vdash_{\mathrm{d}} \texttt{service<}servicePath\texttt{>}\ \ \{properties\}\ \ id\texttt{;} : \tau[id \mapsto t]}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{upa}} userTypePath : userType \quad userType = \mathbf{UserType}(userTypeLabel, \tau_{\mathsf{prop}}) \\ \Gamma,\ \mu,\ \tau, \tau_{\mathsf{prop}} \vdash_{\mathrm{p}} properties \\ \exists userTypeLabel', \mathsf{ProvidedEventType}(userTypeLabel') \in \mu \\ \Delta(userTypeLabel) \sqsubseteq \Delta(userTypeLabel')\end{array}}{\begin{array}{c}\Gamma,\ \mu,\ \tau \vdash_{\mathrm{s}} \texttt{event<}userTypePath\texttt{>}\ \ \{properties\}\ \ id\texttt{;} \\ : \tau[id \mapsto \mathsf{ProvidedEventType}(userTypeLabel)]\end{array}}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{upa}} userTypePath : userType \quad userType = \mathbf{UserType}(userTypeLabel, \tau_{\mathsf{prop}}) \\ \Gamma,\ \mu,\ \tau, \tau_{\mathsf{prop}} \vdash_{\mathrm{p}} properties \\ \exists userTypeLabel', \mathsf{RequiredSessionType}(userTypeLabel') \in \mu \\ \Delta(userTypeLabel) \sqsubseteq \Delta(userTypeLabel')\end{array}}{\begin{array}{c}\Gamma,\ \mu,\ \tau \vdash_{\mathrm{s}} \texttt{session<}userTypePath\texttt{>}\ \ \{properties\}\ \ id\texttt{;} \\ : \tau[id \mapsto \mathsf{RequiredSessionType}(userTypeLabel)]\end{array}}$$

Note that the following rule also apply for `[*]` services instead of `[`i`]`.

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{upa}} userTypePath : userType \\ userType = \mathbf{UserType}(userTypeLabel_1, \tau_{\mathsf{prop}}) \\ \Gamma,\ \mu,\ \tau, \tau_{\mathsf{prop}} \vdash_{\text{p}} properties \\ \tau(service) = \mathbf{ServiceType}(\_, \mu', \_) \qquad i \in \mathbb{N}^* \\ \exists userTypeLabel_2, \mathsf{ProvidedEventType}(userTypeLabel_2) \in \mu' \\ \exists userTypeLabel_3, \mathsf{RequiredEventType}(userTypeLabel_3) \in \mu \\ \Delta(userTypeLabel_2) \sqsubseteq \Delta(userTypeLabel_1) \sqsubseteq \Delta(userTypeLabel_3)\end{array}}{\begin{array}{c}\Gamma,\ \mu,\ \tau \vdash_{\text{d}} \texttt{event<}userTypePath\texttt{> from } service\texttt{[}i\texttt{] \{}properties\texttt{\}}\ id\texttt{;} \\ : \tau[id \mapsto \mathsf{RequiredEventType}(userTypeLabel)]\end{array}}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{upa}} userTypePath : userType \\ userType = \mathbf{UserType}(userTypeLabel_1, \tau_{\mathsf{prop}}) \\ \Gamma,\ \mu,\ \tau, \tau_{\mathsf{prop}} \vdash_{\text{p}} properties \\ \tau(service) = \mathbf{ServiceType}(\_, \mu', \_) \\ \exists userTypeLabel_2, \mathsf{RequiredSessionType}(userTypeLabel_2) \in \mu' \\ \exists userTypeLabel_3, \mathsf{ProvidedSessionType}(userTypeLabel_3) \in \mu \\ \Delta(userTypeLabel_2) \sqsubseteq \Delta(userTypeLabel_1) \sqsubseteq \Delta(userTypeLabel_3)\end{array}}{\begin{array}{c}\Gamma,\ \mu,\ \tau \vdash_{\text{d}} \texttt{session<}userTypePath\texttt{> from } service\texttt{[*] \{}properties\texttt{\}}\ id\texttt{;} \\ : \tau[id \mapsto \mathsf{ProvidedSessionType}(userTypeLabel)]\end{array}}$$

$$\frac{\begin{array}{c}\tau(imType) = \mathsf{RequiredEventType}(t) \\ t' = \mathsf{EventInstType}(t) \\ \Gamma,\ \mu,\ \tau[im \mapsto t'],\ \mathsf{Void} \vdash_{\text{s}} cmpd : \mathsf{Void}\end{array}}{\begin{array}{c}\Gamma,\ \mu,\ \tau \vdash_{\text{d}} \texttt{onReceive}\ id\ \texttt{(}imType\ im\texttt{)}\ cmpd \\ : \tau[id \mapsto \mathsf{EventReceptionType}(imType)]\end{array}}$$

$$\frac{\begin{array}{c}\tau(imType) = \mathsf{ProvidedSessionType}(t) \\ t' = \mathsf{SessionInstType}(t) \\ \Gamma,\ \mu,\ \tau[im \mapsto t'],\ \mathsf{Void} \vdash_{\text{s}} cmpd : \mathsf{Void}\end{array}}{\begin{array}{c}\Gamma,\ \mu,\ \tau \vdash_{\text{d}} \texttt{onReceive}\ id\texttt{(}imType\ im\texttt{)}\ cmpd \\ : \tau[id \mapsto \mathsf{SessionReceptionType}(imType)]\end{array}}$$

The judgments for methods rules are defined as follows :

$$\frac{\Gamma,\ \mu,\ \tau,\ \mathsf{Void} \vdash_{\text{s}} cmpd : \mathsf{Void}}{\Gamma,\ \mu,\ \tau \vdash_{\text{m}} \texttt{initial}\ cmpd} \qquad \frac{\Gamma,\ \mu,\ \tau,\ \mathsf{Void} \vdash_{\text{s}} cmpd : \mathsf{Void}}{\Gamma,\ \mu,\ \tau \vdash_{\text{m}} \texttt{final}\ cmpd} \qquad \frac{\Gamma,\ \mu,\ \tau,\ \mathsf{Void} \vdash_{\text{s}} cmpd : \mathsf{Void}}{\Gamma,\ \mu,\ \tau \vdash_{\text{m}} \texttt{onError}\ cmpd}$$

$$\frac{\begin{array}{c}\Gamma = (\Sigma, \Xi, \Delta) \\ \Gamma,\ \mu,\ \tau \vdash_{\text{t}} type_i : t_i \qquad t_i = \mathsf{DataType}(\_), \forall i \in \{1, \ldots, n\} \\ \Xi^{-1}(cmdName, \langle t_1; \ldots; t_n \rangle) = \mathbf{Command}(cmdLabel, methods) \\ methods(cmdName, \langle t_1; \ldots; t_n \rangle) = ret\_type \qquad \Gamma,\ \mu,\ \tau \vdash_{\text{t}} type : t_d \\ \Gamma,\ \mu,\ \tau[id_1 \mapsto t_1, \ldots, id_n \mapsto t_n],\ ret\_type \vdash_{\text{s}} cmpd : t_s \\ t_d = ret\_type \qquad t_s \sqsubseteq ret\_type \qquad \mathsf{ProvidedCommandType}(cmdLabel) \in \mu\end{array}}{\Gamma,\ \mu,\ \tau \vdash_{\text{m}} \texttt{command}\ type\ cmdName\ \texttt{(}type_1\ id_1\texttt{, ..., }type_n\ id_n\texttt{)}\ cmpd}$$

The judgments for programs rules are defined as follows :

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{spa}} \mathit{servicePath} : \mathbf{ServiceType}(\_, \mu, \tau) \\ \Gamma,\ \mu,\ \tau \vdash_{\text{d}} \mathit{decl}_1 \ldots \mathit{decl}_n : \tau' \\ \Gamma,\ \mu,\ \tau' \vdash_{\text{m}} \mathit{initial} \qquad \Gamma,\ \mu,\ \tau' \vdash_{\text{m}} \mathit{final} \qquad \Gamma,\ \mu,\ \tau' \vdash_{\text{m}} \mathit{error}\end{array}}{\Gamma \vdash_{\text{pg}} \text{'}\mathit{uri\_name}\text{'}\ \texttt{instantiates}\ \mathit{servicePath}\ \{\mathit{decl}_1\ \ldots\ \mathit{decl}_n\ \mathit{initial}\ \mathit{final}\ \mathit{error}\}}$$

D.5 Dynamic Semantics

The global state, σ, describes the state of the current service and its relationships with other services. We assume `onReceive` declarations have been made global with a unique identifier. So, we can ensure to find behavior declarations in σ.

Notations

- Id_{SRV} refers to service identifier
- Id_{M} refers to message identifier
- Id_{S} refers to session identifier
- Id_{E} refers to event identifier
- Id_{BB} refers to bridge behavior identifier
- Id_{D} refers to user datatype identifier

Some Useful Structures and Functions

- $\mathsf{Event} : \mathsf{Id}_D \times \mathsf{properties}$
- $\mathsf{EventMsg} : \mathsf{Id}_{SRV} \times \mathsf{Event}$
- $\mathsf{SessionStatus} : \mathsf{pending} + \mathsf{alive} + \mathsf{dead}$
- $\mathsf{Reception} : \mathsf{Id}_D \times \mathsf{param} \times \mathsf{stmts}$
- $\mathsf{RequiredEvent} : \mathsf{Id}_E \times \mathsf{Service} \times \mathsf{Env}$
- $\mathsf{Service} : \mathsf{Id}_{SRV} \times \mathbb{N} \times (\mathbb{N} \cup \{\top\}) \times \mathsf{Env}$
- $\mathsf{ServiceInst} : \mathbb{N} \times (\mathbb{N} \cup \{\top\}) \times \mathsf{Id}_{SRV}\ \mathsf{list}$
- $\mathsf{Session} : \mathsf{Id}_{SRV} \times \mathsf{Id}_S \times \mathsf{SessionStatus} \times \mathsf{properties}$
- $\mathsf{RequiredSession} : \mathsf{Id}_D \times \mathsf{Env}$
- $\mathsf{ProvidedSession} : \mathsf{Id}_D \times \mathsf{Service} \times \mathsf{Env}$
- $\mathsf{Bridge} : \mathsf{Id}_D \times \mathit{stmts}$
- $\mathsf{BridgeInst} : \mathsf{Id}_{BB} \times \mathsf{Id}_S \times \mathsf{Id}_S$
- $\mathsf{handlers} : \mathsf{handler} \rightarrow (\mathsf{param} \times \mathsf{stmts})$

handler allow to identify a handler, *e.g.*, initial, final, onError or cmd(Id, DataType list)

$$\frac{s = \mathsf{Session}(peer, id_s, \_, properties)}{\mathsf{kill\_session}(s) = \mathsf{Session}(peer, id_s, \mathsf{dead}, properties)}$$

$$\frac{s = \mathsf{Session}(peer, id_s, \mathsf{pending}, properties)}{\mathsf{activate\_session}(s) = \mathsf{Session}(peer, id_s, \mathsf{alive}, properties)}$$

$$\frac{s = \mathsf{Session}(\_, \_, \mathsf{alive}, \_)}{\mathsf{is\_alive}(s) = \mathsf{true}} \qquad \frac{s = \mathsf{Session}(\_, \_, \mathsf{pending}, \_) \vee s = \mathsf{Session}(\_, \_, \mathsf{dead}, \_)}{\mathsf{is\_alive}(s) = \mathsf{false}}$$

The $\triangledown$ function returns the mininum of two values. $\triangledown : \{\mathsf{value} \cup \{\top\}\}_\perp \times \mathsf{value}_\perp \rightarrow \mathsf{value}_\perp$
- If the first argument is undefined (*i.e.*, $\top$), $\triangledown$ returns the value of second argument.
- If a value is an error (*i.e.*, $\perp$), $\triangledown$ returns $\perp$.

Judgment Forms

The judgments for language constructs have the following form, where Γ represents the domain, ϕ the static part of the program, σ the dynamic state of the program, ρ the current variable environment and $t \in ty$.

- $\Gamma, \phi, \sigma, \rho, \mathsf{value} \models_{\mathrm{pl}} place : \sigma', \rho', \mathsf{result}$
- $\Gamma, \phi, \sigma, \rho \models_{\mathrm{s}} stmt : \sigma', \rho', \mathsf{result}$
- $\Gamma, \phi, \sigma, \rho \models_{\mathrm{e}} exp : \sigma', \mathsf{value}$
- $\Gamma, \phi, \sigma, \rho \models_{\mathrm{ps}} properties : (identifier \rightarrow (\sigma \times \mathsf{value})$
- $\Gamma, \phi, \sigma, \rho, (identifier \rightarrow t) \models_{\mathrm{p}} property : (identifier \rightarrow \mathsf{value})$
- $\Gamma, \phi, \sigma, \rho \models_{\mathrm{s}} decl : \sigma', \rho', \mathsf{result}$
- $\Gamma \models_{\mathrm{ph}} path : \mathsf{value}$
- $\mathsf{value\ list} \models_{\mathrm{pa}} params : \rho$

Virtual Machine API

The Pantaxou virtual machine, presented here, is platform independant and is built on top of an implementation-dependant virtual machine. Such a virtual machine is given in section D.6 for the case of a SIP implementation. The functions of the Pantaxou virtual machine are the following.

- $\mathsf{allocate} : \Gamma \times \mathsf{state} \times ty \rightarrow (\mathsf{state} \times \mathsf{value})$
  This function is used to allocate memory for user objects.
- $\mathsf{register} : \mathsf{Id} \times \mathsf{Id}_{SRV} \rightarrow \mathsf{bool}$
  This function is used to register the current service.
- $\mathsf{unregister} : \mathsf{Id} \times \mathsf{Id}_{SRV} \rightarrow \mathsf{unit}$
  This function is used to unregister the current service.
- $\mathsf{discover} : \mathsf{Service} \times (\mathsf{Id} \rightarrow \mathsf{value}) \rightarrow \mathsf{ServiceInst}$
  This function is used to discover available services. The discovery process is based on a service declaration and datatype properties of the requested interaction mode.
- $\mathsf{message} : \mathsf{Id} \times \mathsf{value\ list} \times \mathbb{N} \times \mathbb{N} \times \mathsf{Id}_{SRV}\ \mathsf{list} \rightarrow (\mathsf{Bool} \times \mathsf{value\ list})$
  The function is used to perform remote method invocation between service. It corresponds to the command interaction mode in Pantaxou. It is also used to access service fields. The parameters are the following, field/command name, field value/command arguments, minimum of affected services, number of service requested, list of discovered services to use.
- $\mathsf{subscribe} : \mathsf{Id}_D \times \mathbb{N} \times \mathbb{N} \times \mathsf{Id}_{SRV}\ \mathsf{list} \rightarrow (\mathsf{Bool} \times \mathsf{Id}_{SRV}\ \mathsf{list})$
  This function is used to subscribe to a set of remote services for a particular kind of value type. The parameters are the following, the datatype used for the event, minimum number of services the service subscribes to, the number of services the service requests and the list of discovered services to use.
- $\mathsf{unsubscribe} : \mathsf{Id}_D \times \mathsf{Id}_{SRV}\ \mathsf{list} \rightarrow \mathsf{unit}$
  This function is used to unsubscribe to the set of remote services to which the service has previously subscribe for a particular event type.
- $\mathsf{publish} : \mathsf{state} \times \mathsf{value} \rightarrow \{\top\}_\perp$
  This function is used to publish events to subscribers.
- $\mathsf{invite} : \mathsf{state} \times \mathsf{value} \times \mathbb{N} \times \mathbb{N} \times \mathsf{Id}_{SRV}\ \mathsf{list} \rightarrow (\mathsf{state} \times \mathsf{Bool} \times \mathsf{value\ list})$
  This function is used to initiate a session (value) with remote services. A minimum number of services is specified as well as the number of requested services.
- $\mathsf{decline} : \mathsf{Id}_S \rightarrow \mathsf{unit}$
  This function is used to decline an incoming session.
- $\mathsf{accept} : \mathsf{Id}_S \rightarrow \{\top\}_\perp$
  This function is used to accept an incoming session. It returns $\top$ if the session has been acknowledge, otherwise, it returns $\perp$.
- $\mathsf{bye} : \mathsf{Session} \rightarrow \mathsf{unit}$
  This function is used to close an open session.

In addition to these functions, the virtual machine also provides functions to acknowledge and notify remote service.

- $\mathsf{message}_{try/error} : \mathsf{id}_{message} \rightarrow \mathsf{unit}$
- $\mathsf{message}_{ok} : \mathsf{id}_{message} \times \mathsf{value} \rightarrow \mathsf{unit}$
- $\mathsf{notify}_{try/error/ok} : \mathsf{id}_{event} \rightarrow \mathsf{unit}$
- $\mathsf{invite}_{try} : \mathsf{id}_{session} \rightarrow \mathsf{unit}$
- $\mathsf{bye}_{ok} : \mathsf{id}_{session} \rightarrow \mathsf{unit}$

Auxiliary Functions

The following functions are called internally by the Pantaxou interpreter/compiler. Their function is to easy σ handling. σ stores information about the currently evaluated service, variable environment at the service scope level, active subscriptions, active behaviors, active sessions and active bridges.

state : ((URI × Service) × Env × Subs × Behs × Sessions × Bridges)

- lookup_self : state → (URI × Service)
- lookup_decls : program → decl list
- lookup_env : state → (var → $\text{value}_\perp$)
- lookup_subs : state → (Id_E → Service list)
- lookup_beh : state → (Id_D → value)
- lookup_sessions : state → (Id_S → Session)
- lookup_bridges : state → (Id_S → $\text{BridgeInst}_\perp$)
- lookup : state × Env × Id → value
- clean_session : Γ × program × state × Id_S → state
- clean_all_subscriptions : state → state
- clean_subscriptions : state × Identifier → state
- create_self : Γ × program → state
- lookup_handlers : program → handlers
- update_env : state × var × $\text{value}_\perp$ → state
- update_subs : state × Id_E × Service list → state
- update_beh : state × Id_S × value → state
- update_sessions : state × Id_S × Session → state
- update_bridges : state × Id_S × $\text{BridgeInst}_\perp$ → state
- update : state × Env × Id × value → (state × Env)
- close_session : Γ × program × state × Id_S → state
- close_sessions : Γ × program × state × Id_S list → state
- close_bridge : Γ × program × state × BridgeInst → state

$$\frac{\sigma = \langle self, \_, \_, \_, \_, \_ \rangle}{\text{lookup\_self}(\sigma) = self}$$

$$\frac{\phi_{\text{srvName}} = name \quad \Gamma \models_{\text{ph}} \phi_{\text{srvPath}} : srv \quad \langle (name, srv), \varnothing, \varnothing, \varnothing, \varnothing, \varnothing \rangle = \sigma}{\text{create\_self}(\Gamma, \phi) = \sigma}$$

$$\frac{}{\text{lookup\_decls}(\phi) = \phi_{\text{decls}}}$$

$$\frac{}{\text{lookup\_handlers}(\phi) = \phi_{\text{handlers}}}$$

$$\frac{\sigma = \langle \_, env, \_, \_, \_, \_ \rangle}{\text{lookup\_env}(\sigma) = env}$$

$$\frac{\sigma = \langle \_, \_, subs, \_, \_, \_ \rangle}{\text{lookup\_subs}(\sigma) = subs}$$

$$\frac{\sigma = \langle \_, \_, \_, behs, \_, \_ \rangle}{\text{lookup\_beh}(\sigma) = behs}$$

$$\frac{\sigma = \langle \_, \_, \_, \_, sessions, \_ \rangle}{\text{lookup\_sessions}(\sigma) = sessions}$$

$$\frac{\sigma = \langle \_, \_, \_, \_, \_, bridges \rangle}{\text{lookup\_bridges}(\sigma) = bridges}$$

$$\frac{\sigma = \langle self, env, subs, behs, sessions, bridges\rangle}{\mathsf{update\_env}(\sigma, x, v) = \langle self, env[x \mapsto v], subs, behs, sessions, bridges\rangle}$$

$$\frac{\sigma = \langle self, env, subs, behs, sessions, bridges\rangle}{\mathsf{update\_subs}(\sigma, sub, v) = \langle self, env, subs[sub \mapsto v], behs, sessions, bridges\rangle}$$

$$\frac{\sigma = \langle self, env, subs, behs, sessions, bridges\rangle}{\mathsf{update\_beh}(\sigma, beh, v) = \langle self, env, subs, behs[beh \mapsto v], sessions, bridges\rangle}$$

$$\frac{\sigma = \langle self, env, subs, behs, sessions, bridges\rangle}{\mathsf{update\_sessions}(\sigma, s, v) = \langle self, env, subs, behs, sessions[s \mapsto v], bridges\rangle}$$

$$\frac{\sigma = \langle self, env, subs, behs, sessions, bridges\rangle}{\mathsf{update\_bridges}(\sigma, b, v) = \langle self, env, subs, behs, sessions, bridges[b \mapsto v]\rangle}$$

$$\frac{\mathsf{kill\_session}(\mathsf{lookup\_sessions}(\sigma)(s_{id})) = s \quad \mathsf{update\_sessions}(\sigma, s_{id}, s) = \sigma' \quad \mathsf{lookup\_bridges}(\sigma') = bridges \quad s_{id} \notin \mathsf{dom}(bridges) \vee bridges(s_{id}) = \bot}{\mathsf{clean\_session}(\Gamma, \phi, \sigma, s_{id}) = \sigma'}$$

$$\frac{\mathsf{kill\_session}(\mathsf{lookup\_sessions}(\sigma)(s_{id})) = s \quad \mathsf{update\_sessions}(\sigma, s_{id}, s) = \sigma' \quad \mathsf{lookup\_bridges}(\sigma') = bridges \quad s_{id} \in \mathsf{dom}(bridges) \quad bridges(s_{id}) = b \quad b \neq \bot \quad \mathsf{close\_bridge}(\Gamma, \phi, \sigma', b) = \sigma''}{\mathsf{clean\_session}(\Gamma, \phi, \sigma, s_{id}) = \sigma''}$$

$$\frac{\mathsf{is\_alive}(\mathsf{lookup\_sessions}(\sigma)(s_{id})) = \mathsf{true} \quad \mathsf{bye}(s_{id}) \quad \mathsf{clean\_session}(\Gamma, \phi, \sigma, s_{id}) = \sigma'}{\mathsf{close\_session}(\Gamma, \phi, \sigma, s_{id}) = \sigma'}$$

$$\frac{\mathsf{is\_alive}(\mathsf{lookup\_sessions}(\sigma)(s_{id})) = \mathsf{false}}{\mathsf{close\_session}(\Gamma, \phi, \sigma, s_{id}) = \sigma}$$

$$\frac{\mathsf{close\_session}(\Gamma, \phi, \sigma, s_n) = \sigma' \quad \mathsf{close\_sessions}(\Gamma, \phi, \sigma', \langle s_1; \ldots; s_{n-1}\rangle) = \sigma''}{\mathsf{close\_sessions}(\Gamma, \phi, \sigma, \langle s_1; \ldots; s_n\rangle) = \sigma''}$$

$$\frac{}{\mathsf{close\_sessions}(\Gamma, \phi, \sigma, \langle\rangle) = \sigma}$$

$$\frac{x \in \mathsf{dom}(\rho)}{\mathsf{lookup}(\sigma, \rho, x) = \rho(x)}$$

$$\frac{x \notin \mathsf{dom}(\rho)}{\mathsf{lookup}(\sigma, \rho, x) = \mathsf{lookup\_env}(\sigma)(x)}$$

$$\frac{x \in \mathsf{dom}(\rho)}{\mathsf{update}(\sigma, \rho, x, v) = \sigma, \rho[x \mapsto v]}$$

$$\frac{x \notin \mathsf{dom}(\rho)}{\mathsf{update}(\sigma, \rho, x, v) = \mathsf{update\_env}(\sigma, x, v), \rho}$$

$$\frac{\mathsf{lookup\_subs}(\sigma)(id) = \langle srv_{id_1}; \ldots; srv_{id_m}\rangle \quad \mathsf{update\_subs}(\sigma, id, \langle\rangle) = \sigma' \quad \mathsf{unsubscribe}(id, \langle srv_{id_1}; \ldots; srv_{id_m}\rangle)}{\mathsf{clean\_subscriptions}(\sigma, id) = \sigma'}$$

$$\frac{\begin{array}{c}\mathsf{close\_session}(\Gamma, \phi, \sigma, s_{id_1}) = \sigma_1 \quad \mathsf{close\_session}(\Gamma, \phi, \sigma_1, s_{id_2}) = \sigma_2 \\ \mathsf{lookup\_handlers}(\phi) = handlers \quad handlers(beh) = \langle \_, stmts\rangle \\ \Gamma, \phi, \sigma_2, \rho \models_{\mathrm{s}} stmts : \sigma_3, \top\end{array}}{\mathsf{close\_bridge}(\Gamma, \phi, \sigma, \langle beh, s_{id_1}, s_{id_2}\rangle) = \sigma_3}$$

$$\frac{\mathsf{lookup\_subs}(\sigma) = subs \quad \forall s \in \mathsf{dom}(subs), subs(s) = \langle\rangle}{\mathsf{clean\_all\_subscriptions}(\sigma) = \sigma}$$

$$\frac{\begin{array}{c}\mathsf{lookup\_subs}(\sigma) = subs \quad \exists s \in \mathsf{dom}(subs) | subs(s) \neq \langle\rangle \\ \mathsf{clean\_subscriptions}(\sigma, s) = \sigma' \quad \mathsf{clean\_all\_subscriptions}(\sigma') = \sigma''\end{array}}{\mathsf{clean\_all\_subscriptions}(\sigma) = \sigma''}$$

Program Methods

The following functions are called by the underlying virtual machine. They evalute program handlers.

- initial : $\Gamma \times$ program $\rightarrow$ state$_\perp$
- final : $\Gamma \times$ program $\times$ state $\rightarrow$ state
- cmd : $\Gamma \times$ program $\times$ state $\times$ Id$_M \times$ Id $\times$ DataType list $\times$ value list $\rightarrow$ state
- field : $\Gamma \times$ program $\times$ state $\times$ Id$_M \times$ Id $\times$ value list $\rightarrow$ state
- event : $\Gamma \times$ program $\times$ state $\times$ Id$_{SRV} \times$ Id$_E \times ty \times$ value $\rightarrow$ state
- session : $\Gamma \times$ program $\times$ state $\times$ Id$_{SRV} \times$ Id$_M \times ty \times$ value $\rightarrow$ state
- session_close : $\Gamma \times$ program $\times$ state $\times$ Session $\rightarrow$ state
- onError : $\Gamma \times$ program $\times$ state $\rightarrow$ state

$$\frac{\begin{array}{c}\mathsf{create\_self}(\Gamma, \phi) = \sigma \quad \mathsf{register}(\mathsf{lookup\_self}(\sigma)) = \mathsf{true} \\ \mathsf{lookup\_decls}(\phi) = decls \quad \Gamma, \phi, \sigma, \varnothing \models_{\mathrm{s}} decls : \sigma_1, \rho, \top \\ \sigma_1 = \langle self, \_, subs, sessions, bridges\rangle \\ \langle self, \rho, subs, sessions, bridges\rangle = \sigma_2 \\ \mathsf{lookup\_handlers}(\phi) = handlers \quad handlers(\mathsf{initial}) = \langle\langle\rangle, stmts\rangle \\ \Gamma, \phi, \sigma_2, \varnothing \models_{\mathrm{s}} stmts : \sigma_3, \_, \top\end{array}}{\mathsf{initial}(\Gamma, \phi) \Rightarrow \sigma_3}$$

$$\frac{\begin{array}{c}\mathsf{create\_self}(\phi) = \sigma \quad \mathsf{register}(\mathsf{lookup\_self}(\sigma)) = \mathsf{true} \\ \mathsf{lookup\_decls}(\phi) = decls \quad \Gamma, \phi, \sigma, \varnothing \models_{\mathrm{s}} decls : \sigma_1, \rho, \top \\ \sigma_1 = \langle self, \_, subs, sessions, bridges\rangle \\ \langle self, \rho, subs, sessions, bridges\rangle = \sigma_2 \\ \mathsf{lookup\_handlers}(\phi) = handlers \quad handlers(\mathsf{initial}) = \langle\langle\rangle, stmts\rangle \\ \Gamma, \phi, \sigma_2, \varnothing \models_{\mathrm{s}} stmts : \sigma_3, \_, \bot \quad \mathsf{onError}(\Gamma, \phi, \sigma_3) = \sigma_4\end{array}}{\mathsf{initial}(\Gamma, \phi) \Rightarrow \sigma_4}$$

$$\frac{\begin{array}{c}\mathsf{create\_self}(\phi) = \sigma \quad \mathsf{register}(\mathsf{lookup\_self}(\sigma)) = \mathsf{true} \\ \mathsf{lookup\_decls}(\phi) = decls \quad \Gamma, \phi, \sigma, \varnothing \models_{\mathrm{s}} decls : \sigma_1, \rho, \bot \quad \mathsf{onError}(\Gamma, \phi, \sigma_1) = \sigma_2\end{array}}{\mathsf{initial}(\Gamma, \phi) \Rightarrow \sigma_2}$$

$$\frac{\mathsf{create\_self}(\phi) = \sigma \quad \mathsf{register}(\mathsf{lookup\_self}(\sigma)) = \mathsf{false}}{\mathsf{initial}(\Gamma, \phi) \Rightarrow \perp}$$

$$\frac{\begin{array}{c}\mathsf{lookup\_handlers}(\phi) = handlers \\ handlers(\mathsf{onError}) = \langle \langle\rangle, stmts\rangle \\ \Gamma, \phi, \sigma, \varnothing \models_{\mathrm{s}} stmts : \sigma', \_, \top\end{array}}{\mathsf{onError}(\Gamma, \phi, \sigma) \Rightarrow \sigma'}$$

$$\frac{\begin{array}{c}\mathsf{unregister}(\mathsf{lookup\_self}(\sigma)) \\ \mathsf{clean\_all\_subscriptions}(\sigma) = \sigma' \\ \mathsf{lookup\_handlers}(\phi) = handlers \\ handlers(\mathsf{final}) = \langle \langle\rangle, stmts\rangle \\ \Gamma, \phi, \sigma', \varnothing \models_{\mathrm{s}} stmts : \sigma'', \_, \top\end{array}}{\mathsf{final}(\Gamma, \phi, \sigma) \Rightarrow \sigma''}$$

$$\frac{\begin{array}{c}\mathsf{unregister}(\mathsf{lookup\_self}(\sigma)) \\ \mathsf{clean\_all\_subscriptions}(\sigma) = \sigma' \\ \mathsf{lookup\_handlers}(\phi) = handlers \\ handlers(\mathsf{final}) = \langle \langle\rangle, stmts\rangle \\ \Gamma, \phi, \sigma', \varnothing \models_{\mathrm{s}} stmts : \sigma'', \_, \bot \\ \mathsf{onError}(\Gamma, \phi, \sigma'') = \sigma'''\end{array}}{\mathsf{final}(\Gamma, \phi, \sigma) \Rightarrow \sigma'''}$$

$$\frac{\begin{array}{c}\mathsf{notify}_{try}(id_e) \\ \mathsf{lookup\_beh}(\sigma)(t_e) = beh \\ beh = \mathsf{Reception}(\_, param, stmts) \\ \langle \mathsf{EventMsg}(src, v_e)\rangle \models_{\mathrm{pa}} \langle param\rangle : \rho' \\ \Gamma, \phi, \sigma, \rho' \models_{\mathrm{s}} stmts : \sigma', \_, \top \\ \mathsf{notify}_{ok}(id_e)\end{array}}{\mathsf{event}(\Gamma, \phi, \sigma, src, id_e, t_e, v_e) \Rightarrow \sigma'}$$

$$\frac{\begin{array}{c}\mathsf{notify}_{try}(id_e) \\ \mathsf{lookup\_beh}(\sigma)(t_e) = beh \\ beh = \mathsf{Reception}(\_, param, stmts) \\ \langle \mathsf{EventMsg}(src, v_e)\rangle \models_{\mathrm{pa}} \langle param\rangle : \rho' \\ \Gamma, \phi, \sigma, \rho' \models_{\mathrm{s}} stmts : \sigma', \_, \bot \\ \mathsf{onError}(\Gamma, \phi, \sigma') = \sigma'' \\ \mathsf{notify}_{error}(id_e)\end{array}}{\mathsf{event}(\Gamma, \phi, \sigma, src, id_e, t_e, v_e) \Rightarrow \sigma''}$$

$$\frac{\begin{array}{c}\mathsf{message}_{try}(id_m) \\ \mathsf{lookup\_env}(\sigma)(field) = v \\ \mathsf{message}_{ok}(id_m, v)\end{array}}{\mathsf{field}(\Gamma, \phi, \sigma, id_m, field, \langle\rangle) \Rightarrow \sigma}$$

$$\frac{\begin{array}{c}\mathsf{message}_{try}(id_m) \\ \mathsf{update\_env}(\sigma, field, v) = \sigma' \\ \mathsf{message}_{ok}(id_m, v)\end{array}}{\mathsf{field}(\Gamma, \phi, \sigma, id_m, field, \langle v\rangle) \Rightarrow \sigma'}$$

$$\frac{\begin{array}{c}\mathsf{message}_{try}(id_m) \\ \mathsf{lookup\_handlers}(\phi) = handlers \\ handlers(\mathsf{cmd}(mth, \langle t_1; \ldots; t_n\rangle)) = \langle params, stmts\rangle \\ \langle v_1; \ldots; v_n\rangle \models_{\mathrm{pa}} params : \rho' \\ \Gamma, \phi, \sigma, \rho' \models_{\mathrm{s}} stmts : \sigma', \_, r \qquad\qquad r \neq \bot \\ \mathsf{message}_{ok}(id_m, r)\end{array}}{\mathsf{cmd}(\Gamma, \phi, \sigma, id_m, mth, \langle t_1; \ldots; t_n\rangle, \langle v_1; \ldots; v_n\rangle) \Rightarrow \sigma'}$$

$$\frac{\begin{array}{c}\mathsf{message}_{try}(id_m) \\ \mathsf{lookup\_handlers}(\phi) = handlers \\ handlers(\mathsf{cmd}(mth, \langle t_1; \ldots; t_n\rangle)) = \langle params, stmts\rangle \\ \langle v_1; \ldots; v_n\rangle \models_{\mathrm{pa}} params : \rho' \\ \Gamma, \phi, \sigma, \rho' \models_{\mathrm{s}} stmts : \sigma', \_, \bot \\ \mathsf{onError}(\Gamma, \phi, \sigma') = \sigma'' \\ \mathsf{message}_{error}(id_m)\end{array}}{\mathsf{cmd}(\Gamma, \phi, \sigma, id_m, mth, \langle t_1; \ldots; t_n\rangle, \langle v_1; \ldots; v_n\rangle) \Rightarrow \sigma''}$$

$$\frac{\begin{array}{c}\mathsf{invite}_{try}(id_s)\\ \mathsf{lookup\_handlers}(\phi) = handlers\\ \mathsf{lookup\_beh}(\sigma)(t_s) = beh\\ handlers(beh) = \mathsf{Reception}(\_, param, stmts)\\ \langle \mathsf{Session}(src, id_s, \mathsf{pending}, v_s)\rangle \models_{\text{pa}} \langle param\rangle : \rho'\\ \Gamma, \phi, \sigma, \rho' \models_{\text{s}} stmts : \sigma', \_, \top\end{array}}{\mathsf{session}(\Gamma, \phi, \sigma, src, id_s, t_s, v_s) \Rightarrow \sigma'} \qquad \frac{\begin{array}{c}\mathsf{bye\_ok}(s_{id})\\ \mathsf{clean\_session}(\Gamma, \phi, \sigma, s_{id}) = \sigma'\end{array}}{\mathsf{session\_close}(\Gamma, \phi, \sigma, s_{id}) \Rightarrow \sigma'}$$

D.5.1 Semantics of Top Level Declarations and Handler Parameters

For Service Declarations

Service declarations are evaluated with the statement judgments.

For Parameters (Handlers, Methods, Commands)

$$\frac{\begin{array}{c}v_1, \varnothing \models_{\text{pa}} param_1 : \rho_1\\ \ldots\\ v_n, \rho_{n-1} \models_{\text{pa}} param_n : \rho_n\end{array}}{\langle v_1; \ldots; v_n\rangle \models_{\text{pa}} \langle param_1; \ldots; param_n\rangle : \rho_n} \qquad \frac{}{v, \rho \models_{\text{pa}} param : \rho[param \mapsto v]}$$

D.5.2 Semantics of Places

$$\frac{\mathsf{update}(\sigma, \rho, identifier, v) = \sigma', \rho'}{\Gamma, \phi, \sigma, \rho, v \models_{\text{pl}} identifier : \sigma', \rho', \top} \qquad \frac{\Gamma, \phi, \sigma, \rho \models_{\text{e}} exp : \bot}{\Gamma, \phi, \sigma, \rho, v \models_{\text{pl}} exp.field : \sigma, \bot}$$

$$\frac{\begin{array}{c}\Gamma, \phi, \sigma, \rho \models_{\text{e}} exp : \mathsf{ServiceInst}(1, 1, \langle srv_{id_1}; \ldots; srv_{id_p}\rangle)\\ \mathsf{message}(field, \langle v\rangle, 1, 1, \langle srv_{id_1}; \ldots; srv_{id_p}\rangle) =\\ \mathsf{true}, \_\end{array}}{\Gamma, \phi, \sigma, \rho, v \models_{\text{pl}} exp.field : \sigma, \top}$$

$$\frac{\begin{array}{c}\Gamma, \phi, \sigma, \rho \models_{\text{e}} exp : \mathsf{ServiceInst}(1, 1, \langle srv_{id_1}; \ldots; srv_{id_p}\rangle)\\ \mathsf{message}(field, \langle v\rangle, 1, 1, \langle srv_{id_1}; \ldots; srv_{id_p}\rangle) =\\ \mathsf{false}, \_\end{array}}{\Gamma, \phi, \sigma, \rho, v \models_{\text{pl}} exp.field : \sigma, \bot}$$

$$\frac{\begin{array}{c}\Gamma, \phi, \sigma, \rho \models_{\text{e}} exp : \mathsf{ServiceInst}(i, i, \langle srv_{id_1}; \ldots; srv_{id_p}\rangle)\\ \mathsf{message}(field, \langle v\rangle, i, i, \langle srv_{id_1}; \ldots; srv_{id_p}\rangle) =\\ \mathsf{true}, \_\end{array}}{\Gamma, \phi, \sigma, \rho, v \models_{\text{pl}} exp.field : \sigma, \top}$$

$$\frac{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \mathsf{ServiceInst}(i, i, \langle srv_{id_1}; \ldots; srv_{id_p}\rangle) \quad \mathsf{message}(\mathit{field}, \langle v\rangle, i, i, \langle srv_{id_1}; \ldots; srv_{id_p}\rangle) = \mathsf{false}, \_}{\Gamma,\phi,\sigma,\rho,v \models_{\text{pl}} exp\,.\,\mathit{field} : \sigma, \bot}$$

$$\frac{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \mathsf{ServiceInst}(1, \top, \langle srv_{id_1}; \ldots; srv_{id_p}\rangle) \quad \mathsf{message}(\mathit{field}, \langle v\rangle, 1, p, \langle srv_{id_1}; \ldots; srv_{id_p}\rangle) = \mathsf{true}, \_}{\Gamma,\phi,\sigma,\rho,v \models_{\text{pl}} exp\,.\,\mathit{field} : \sigma, \top}$$

$$\frac{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \mathsf{ServiceInst}(1, \top, \langle srv_{id_1}; \ldots; srv_{id_p}\rangle) \quad \mathsf{message}(\mathit{field}, \langle v\rangle, 1, p, \langle srv_{id_1}; \ldots; srv_{id_p}\rangle) = \mathsf{false}, \_}{\Gamma,\phi,\sigma,\rho,v \models_{\text{pl}} exp\,.\,\mathit{field} : \sigma, \bot}$$

D.5.3 Semantics of Statements

Compound

$$\frac{}{\Gamma,\phi,\sigma,\rho \models_{\text{s}} \{\,\} : \sigma, \rho, \top}$$

$$\frac{\Gamma,\phi,\sigma,\rho \models_{\text{s}} stmt_1 : \sigma_1, \rho_1, v_1 \quad \cdots \quad \Gamma,\phi,\sigma_{i-1},\rho_{i-1} \models_{\text{s}} stmt_i : \sigma_i, \rho_i, v_i \quad \forall j < i, v_j = \top \quad (i < n \wedge v_i \neq \top) \bigvee (i = n)}{\Gamma,\phi,\sigma,\rho \models_{\text{s}} \{\, stmt_1 \ \ldots \ stmt_n \,\} : \sigma_i, \rho_i, v_i}$$

$$\frac{\Gamma,\phi,\sigma,\rho \models_{\text{s}} stmt_1 : \sigma_1, \rho_1, v_1 \quad \cdots \quad \Gamma,\phi,\sigma_{i-1},\rho_{i-1} \models_{\text{s}} stmt_i : \sigma_i, \rho_i, v_i \quad \forall j < i, v_j = \top \quad i \leq n \wedge v_i = \bot}{\Gamma,\phi,\sigma,\rho \models_{\text{s}} \{\, stmt_1 \ \ldots \ stmt_n \,\} : \sigma_i, \rho_i, \bot}$$

Void/$\top$ Expression

$$\frac{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \sigma', v \quad v = \top \vee v = \mathsf{Void} \vee v = \mathsf{List}(\mathsf{Void})}{\Gamma,\phi,\sigma,\rho \models_{\text{s}} exp\ \texttt{;} : \sigma', \rho, \top}$$

$$\frac{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \sigma', \bot}{\Gamma,\phi,\sigma,\rho \models_{\text{s}} exp\ \texttt{;} : \sigma', \rho, \bot}$$

Assignment

$$\frac{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \sigma', v \quad v \neq \bot \quad \Gamma,\phi,\sigma',\rho,v \models_{\text{pl}} place : \sigma'', \rho', r}{\Gamma,\phi,\sigma,\rho \models_{\text{s}} place\ =\ exp\ \texttt{;} : \sigma'', \rho', r}$$

$$\frac{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \sigma', \bot}{\Gamma,\phi,\sigma,\rho \models_{\text{s}} place\ =\ exp\ \texttt{;} : \sigma', \rho, \bot}$$

Return

$$\frac{}{\Gamma, \phi, \sigma, \rho \models_{\text{s}} \texttt{return ;} : \sigma, \rho, \mathsf{Void}} \qquad \frac{\Gamma, \phi, \sigma, \rho \models_{\text{e}} exp : \sigma', v}{\Gamma, \phi, \sigma, \rho \models_{\text{s}} \texttt{return } exp \texttt{ ;} : \sigma', \rho, v}$$

If

$$\frac{\Gamma, \phi, \sigma, \rho \models_{\text{e}} exp : \sigma', \mathsf{true} \qquad \Gamma, \phi, \sigma', \rho \models_{\text{s}} cmpd_1 : \sigma'', \rho', r}{\Gamma, \phi, \sigma, \rho \models_{\text{s}} \texttt{if(}exp\texttt{)} \;\; cmpd_1 \;\; \texttt{else} \;\; cmpd_2 : \sigma'', \rho', r}$$

$$\frac{\Gamma, \phi, \sigma, \rho \models_{\text{e}} exp : \sigma', \mathsf{false} \qquad \Gamma, \phi, \sigma', \rho \models_{\text{s}} cmpd_2 : \sigma'', \rho', r}{\Gamma, \phi, \sigma, \rho \models_{\text{s}} \texttt{if(}exp\texttt{)} \;\; cmpd_1 \;\; \texttt{else} \;\; cmpd_2 : \sigma'', \rho', r}$$

Reject

$$\frac{\begin{array}{c}\Gamma, \phi, \sigma, \rho \models_{\text{e}} exp_1 : \sigma_1, s_{id} \\ s_{id} \neq \bot \\ \mathsf{decline}(s_{id}) \\ \mathsf{clean\_session}(\Gamma, \phi, \sigma_1, s_{id}) = \sigma_2\end{array}}{\Gamma, \phi, \sigma, \rho \models_{\text{s}} \texttt{reject (}exp_1\texttt{) ;} : \sigma_2, \rho, \top} \qquad \frac{\Gamma, \phi, \sigma, \rho \models_{\text{e}} exp_1 : \sigma', \bot}{\Gamma, \phi, \sigma, \rho \models_{\text{s}} \texttt{reject (}exp_1\texttt{) ;} : \sigma', \rho, \bot}$$

Adopt

For event

$$\frac{\begin{array}{c}\Gamma, \phi, \sigma, \rho \models_{\text{e}} exp : \sigma_1, r \qquad r = \mathsf{Reception}(id, \_, \_) \\ \mathsf{lookup}(\sigma_1, \rho, id) = \mathsf{RequiredEvent}(label, service, properties) \\ service = Service(\_, count\_min, count\_rq, \_) \\ \mathsf{discover}(service, properties) = \langle srv_{id_1}; \ldots; srv_{id_n} \rangle \\ m = count\_rq \triangledown n \qquad count\_min \leq m \\ \mathsf{clean\_subscriptions}(\sigma_1, id) = \sigma_2 \\ \mathsf{subscribe}(label, count\_min, m, \langle srv_{id_1}; \ldots; srv_{id_n} \rangle) = \mathsf{true}, srvList \\ \mathsf{update\_subs}(\sigma_2, id, srvList) = \sigma_3 \\ \mathsf{update\_beh}(\sigma_3, label, r) = \sigma_4\end{array}}{\Gamma, \phi, \sigma, \rho \models_{\text{s}} \texttt{adopt (}exp\texttt{) ;} : \sigma_4, \rho, \top}$$

$$\frac{\begin{array}{c}\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \sigma_1, \mathsf{Reception}(id, \_, \_) \\ \mathsf{lookup}(\sigma_1, \rho, id) = \mathsf{RequiredEvent}(label, service, properties) \\ service = Service(\_, count\_min, count\_rq, \_) \\ \mathsf{discover}(service, properties) = \langle srv_{id_1}; \ldots; srv_{id_n} \rangle \\ m = count\_rq \nabla n \qquad count\_min \leq m \\ \mathsf{clean\_subscriptions}(\sigma_1, id) = \sigma_2 \\ \mathsf{subscribe}(label, count\_min, m, \langle srv_{id_1}; \ldots; srv_{id_n} \rangle) = \mathsf{false}, srvList \\ \mathsf{update\_subs}(\sigma_2, id, srvList) = \sigma_3 \\ \mathsf{clean\_subscriptions}(\sigma_3, id) = \sigma_4 \end{array}}{\Gamma,\phi,\sigma,\rho \models_{\text{s}} \texttt{adopt (}exp\texttt{) ; :} \sigma_4, \rho, \bot}$$

$$\frac{\begin{array}{c}\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \sigma', \mathsf{Reception}(id, \_, \_) \\ \mathsf{lookup}(\sigma', \rho, id) = \mathsf{RequiredEvent}(label, service, properties) \\ service = Service(\_, count\_min, count\_rq, \_) \\ \mathsf{discover}(service, properties) = \langle srv_{id_1}; \ldots; srv_{id_n} \rangle \\ m = count\_rq \nabla n \qquad count\_min > m \end{array}}{\Gamma,\phi,\sigma,\rho \models_{\text{s}} \texttt{adopt (}exp\texttt{) ; :} \sigma', \rho, \bot}$$

For session

$$\frac{\begin{array}{c}\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \sigma', \mathsf{Reception}(id, \_, \_) \\ \mathsf{lookup}(\sigma', \rho, id) = \mathsf{ProvidedSession}(label, service, properties) \\ \mathsf{discover}(service, properties) = \langle srv_{id_1}; \ldots; srv_{id_n} \rangle \\ n > 0 \\ \mathsf{update\_beh}(\sigma', label, beh) = \sigma'' \end{array}}{\Gamma,\phi,\sigma,\rho \models_{\text{s}} \texttt{adopt (}exp\texttt{) ; :} \sigma'', \rho, \top}$$

$$\frac{\begin{array}{c}\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \sigma', \mathsf{Reception}(id, \_, \_) \\ \mathsf{lookup}(\sigma', \rho, id) = \mathsf{ProvidedSession}(label, service, properties) \\ \mathsf{discover}(service, properties) = \langle srv_{id_1}; \ldots; srv_{id_n} \rangle \\ n = 0 \end{array}}{\Gamma,\phi,\sigma,\rho \models_{\text{s}} \texttt{adopt (}exp\texttt{) ; :} \sigma', \rho, \bot}$$

$$\frac{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \sigma', \bot}{\Gamma,\phi,\sigma,\rho \models_{\text{s}} \texttt{adopt (}exp\texttt{) ; :} \sigma', \rho, \bot}$$

Build-in Methods

$$\frac{\begin{array}{c}\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \sigma', s \\ \mathsf{close\_session}(\Gamma, \phi, \sigma', s) = \sigma'' \end{array}}{\Gamma,\phi,\sigma,\rho \models_{\text{s}} \texttt{disconnect (}exp\texttt{) ; :} \sigma'', \rho, \top}$$

$$\frac{\begin{array}{c}\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \sigma', \mathsf{BridgeInst}(Id_{BB}, Id_{S_1}, Id_{S_2}) \\ \mathsf{close\_bridge}(\Gamma, \phi, \sigma', \langle Id_{BB}, Id_{S_1}, Id_{S_2} \rangle) = \sigma'' \end{array}}{\Gamma,\phi,\sigma,\rho \models_{\text{s}} \texttt{disconnect (}exp\texttt{) ; :} \sigma'', \rho, \top}$$

$$\frac{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \sigma', \bot}{\Gamma,\phi,\sigma,\rho \models_{\text{s}} \texttt{disconnect (}exp\texttt{) ; :} \sigma', \rho, \bot}$$

Invite

$$\frac{\begin{array}{c}\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp_{srv} : \sigma_1, srv \qquad srv = \mathsf{Service}(\_, 1, 1, \_) \\ \Gamma,\phi,\sigma_1,\rho \models_{\text{e}} exp_{sess} : \sigma_2, \mathsf{SessionInst}(sesslabel, sessProperties) \\ \mathsf{discover}(srv, sessProperties) = \langle srv_1; \ldots; srv_n\rangle \\ n > 0 \\ \mathsf{invite}(\sigma_2, s, 1, 1, \langle srv_1; \ldots; srv_n\rangle) = \sigma_3, \mathsf{true}, \langle v_1\rangle \\ \Gamma,\phi,\sigma_3,\rho[id \mapsto v_1] \models_{\text{s}} cmpd_{acc} : \sigma_4, \rho', v\end{array}}{\begin{array}{r}\Gamma,\phi,\sigma,\rho \models_{\text{s}} \texttt{invite (}exp_{srv}\texttt{, } exp_{sess}\texttt{) \{accepted(}type\ id\texttt{) } cmpd_{acc} \\ \texttt{rejected() } cmpd_{rej} \texttt{ }\} : \sigma_4, \rho', v\end{array}}$$

$$\frac{\begin{array}{c}\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp_{srv} : \sigma_1, srv \qquad srv = \mathsf{Service}(\_, 1, 1, \_) \\ \Gamma,\phi,\sigma_1,\rho \models_{\text{e}} exp_{sess} : \sigma_2, \mathsf{SessionInst}(sesslabel, sessProperties) \\ \mathsf{discover}(srv, sessProperties) = \langle srv_1; \ldots; srv_n\rangle \\ n > 0 \\ \mathsf{invite}(\sigma_2, s, 1, 1, \langle srv_1; \ldots; srv_n\rangle) = \sigma_3, \mathsf{false}, \_ \\ \Gamma,\phi,\sigma_3,\rho \models_{\text{s}} cmpd_{rej} : \sigma_4, \rho', v\end{array}}{\begin{array}{r}\Gamma,\phi,\sigma,\rho \models_{\text{s}} \texttt{invite (}exp_{srv}\texttt{, } exp_{sess}\texttt{) \{accepted(}type\ id\texttt{) } cmpd_{acc} \\ \texttt{rejected() } cmpd_{rej} \texttt{ }\} : \sigma_4, \rho', v\end{array}}$$

$$\frac{\begin{array}{c}\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp_{srv} : \sigma_1, srv \qquad srv = \mathsf{Service}(\_, count\_min, count\_rq, \_) \\ \Gamma,\phi,\sigma_1,\rho \models_{\text{e}} exp_{sess} : \sigma_2, \mathsf{RequiredSession}(sesslabel, sessProperties) \\ \mathsf{discover}(srv, sessProperties) = \langle srv_1; \ldots; srv_n\rangle \\ m = count\_rq \nabla n \qquad count\_min \leq m \\ \mathsf{invite}(\sigma_2, s, count\_min, m, \langle srv_1; \ldots; srv_n\rangle) = \sigma_3, \mathsf{true}, \langle v_1; \ldots; v_i\rangle \\ \Gamma,\phi,\sigma_3,\rho[id \mapsto \langle v_1; \ldots; v_i\rangle] \models_{\text{s}} cmpd_{acc} : \sigma_4, \rho', v\end{array}}{\begin{array}{r}\Gamma,\phi,\sigma,\rho \models_{\text{s}} \texttt{invite (}exp_{srv}\texttt{, } exp_{sess}\texttt{) \{accepted(}type\ id\texttt{) } cmpd_{acc} \\ \texttt{rejected() } cmpd_{rej} \texttt{ }\} : \sigma_4, \rho', v\end{array}}$$

$$\frac{\begin{array}{c}\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp_{srv} : \sigma_1, srv \qquad srv = \mathsf{Service}(\_, count\_min, count\_rq, \_) \\ \Gamma,\phi,\sigma_1,\rho \models_{\text{e}} exp_{sess} : \sigma_2, \mathsf{RequiredSession}(sesslabel, sessProperties) \\ \mathsf{discover}(srv, sessProperties) = \langle srv_1; \ldots; srv_n\rangle \\ m = count\_rq \nabla n \qquad count\_min \leq m \\ \mathsf{invite}(\sigma_2, s, count\_min, m, \langle srv_1; \ldots; srv_n\rangle) = \sigma_3, \mathsf{false}, \langle v_1; \ldots; v_i\rangle \\ \mathsf{close\_sessions}(\Gamma,\phi,\sigma_3,\langle v_1; \ldots; v_i\rangle) = \sigma_4 \\ \Gamma,\phi,\sigma_4,\rho \models_{\text{s}} cmpd_{rej} : \sigma_5, \rho', v\end{array}}{\begin{array}{r}\Gamma,\phi,\sigma,\rho \models_{\text{s}} \texttt{invite (}exp_{srv}\texttt{, } exp_{sess}\texttt{) \{accepted(}type\ id\texttt{) } cmpd_{acc} \\ \texttt{rejected() } cmpd_{rej} \texttt{ }\} : \sigma_5, \rho', v\end{array}}$$

$$\frac{\begin{array}{c}\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp_{srv} : \sigma_1, srv \qquad srv = \mathsf{Service}(\_, count\_min, count\_rq, \_) \\ \Gamma,\phi,\sigma_1,\rho \models_{\text{e}} exp_{sess} : \sigma_2, \mathsf{RequiredSession}(sesslabel, sessProperties) \\ \mathsf{discover}(srv, sessProperties) = \langle srv_1; \ldots; srv_n \rangle \\ m = count\_rq \nabla n \qquad count\_min > m\end{array}}{\begin{array}{r}\Gamma,\phi,\sigma,\rho \models_{\text{s}} \texttt{invite (}exp_{srv}\texttt{,}\ exp_{sess}\texttt{)\{accepted(}type\ id\texttt{)}\ \ cmpd_{acc} \\ \texttt{rejected()}\ \ cmpd_{rej}\ \texttt{\}} : \sigma_3, \rho', \bot\end{array}}$$

Publish

$$\frac{\begin{array}{c}\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \sigma_1, v \qquad v \neq \bot \\ \mathsf{publish}(\sigma_1, v) = r\end{array}}{\Gamma,\phi,\sigma,\rho \models_{\text{s}} \texttt{publish (}exp\texttt{) ;} : \sigma_1, \rho, r} \qquad \frac{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \sigma_1, \bot}{\Gamma,\phi,\sigma,\rho \models_{\text{s}} \texttt{publish (}exp\texttt{) ;} : \sigma_1, \rho, \bot}$$

D.5.4 Semantics of Expressions

Constant

$$\frac{c \in \{\texttt{"..."}, \ldots\}}{\Gamma,\phi,\sigma,\rho \models_{\text{e}} c : \sigma, \mathsf{String}(c)} \qquad \frac{c \in \mathbb{Z}}{\Gamma,\phi,\sigma,\rho \models_{\text{e}} c : \sigma, \mathsf{Int}(c)}$$

$$\frac{c = \texttt{true}}{\Gamma,\phi,\sigma,\rho \models_{\text{e}} c : \sigma, \mathsf{Bool}(\mathsf{true})} \qquad \frac{c = \texttt{false}}{\Gamma,\phi,\sigma,\rho \models_{\text{e}} c : \sigma, \mathsf{Bool}(\mathsf{false})}$$

$$\frac{c \in \{\texttt{'...'}, \ldots\}}{\Gamma,\phi,\sigma,\rho \models_{\text{e}} c : \sigma, \mathsf{URI}(c)}$$

Identifier

$$\frac{\mathsf{lookup}(\sigma, \rho, identifier) = v}{\Gamma,\phi,\sigma,\rho \models_{\text{e}} identifier : \sigma, v}$$

Constructor

$$\frac{\mathsf{allocate}(\Gamma, \sigma, type) = \langle \sigma', v \rangle}{\Gamma,\phi,\sigma,\rho \models_{\text{e}} \texttt{new}\ \ type : \sigma', v}$$

$$\frac{\begin{array}{c}\mathsf{discover}(\rho(srv), \varnothing) = \langle srv_1; \ldots; srv_p \rangle \\ p < i\end{array}}{\Gamma,\phi,\sigma,\rho \models_{\text{e}} \texttt{new}\ \ service\texttt{[i]} : \sigma, \bot} \qquad \frac{\mathsf{discover}(\rho(srv), \varnothing) = \langle\rangle}{\Gamma,\phi,\sigma,\rho \models_{\text{e}} \texttt{new}\ \ service\texttt{[*]} : \sigma, \bot}$$

$$\frac{\begin{array}{c}\mathsf{discover}(\rho(srv), \varnothing) = \langle srv_1; \ldots; srv_p \rangle \\ 1 \leq i \leq p\end{array}}{\Gamma,\phi,\sigma,\rho \models_{\text{e}} \texttt{new}\ \ service\texttt{[i]} : \sigma, \mathsf{ServiceInst}(i, i, \langle srv_1; \ldots; srv_p \rangle)}$$

$$\frac{\mathsf{discover}(\rho(srv), \varnothing) = \langle srv_1; \ldots; srv_p \rangle \quad p > 0}{\Gamma, \phi, \sigma, \rho \models_{\mathrm{e}} \texttt{new}\ \ service\texttt{[}\star\texttt{]} : \sigma, \mathsf{ServiceInst}(1, \top, \langle srv_1; \ldots; srv_p \rangle)}$$

Call

$$\frac{\begin{array}{c}\Gamma, \phi, \sigma, \rho \models_{\mathrm{e}} exp_1 : \sigma_1, v_1 \\ \ldots \\ \Gamma, \phi, \sigma_{n-1}, \rho \models_{\mathrm{e}} exp_n : \sigma_n, v_n \\ \mathsf{lookup\_handlers}(\phi)(fct) = \langle params, decls, stmts \rangle \\ \langle v_1; \ldots; v_n \rangle \models_{\mathrm{pa}} params : \rho' \qquad \forall i \in [1; \ldots; n], v_i \neq \bot \\ \Gamma, \phi, \sigma_n, \rho' \models_{\mathrm{s}} decls, stmts : \sigma', \_, v\end{array}}{\Gamma, \phi, \sigma, \rho \models_{\mathrm{e}} fct\texttt{(}exp_1\texttt{,}\ \ldots\ \texttt{,}\ exp_n\texttt{)} : \sigma', v}$$

$$\frac{\begin{array}{c}\Gamma, \phi, \sigma, \rho \models_{\mathrm{e}} exp_1 : \sigma_1, v_1 \\ \ldots \\ \Gamma, \phi, \sigma_{n-1}, \rho \models_{\mathrm{e}} exp_n : \sigma_n, v_n \\ \exists i \in [1; \ldots; n], v_i = \bot\end{array}}{\Gamma, \phi, \sigma, \rho \models_{\mathrm{e}} fct\texttt{(}exp_1\texttt{,}\ \ldots\ \texttt{,}\ exp_n\texttt{)} : \sigma_n, \bot}$$

Accept

$$\frac{\begin{array}{c}\Gamma, \phi, \sigma, \rho \models_{\mathrm{e}} exp : \sigma', s_{id} \\ \mathsf{accept}(s_{id}) = \top \\ \mathsf{activate\_session}(\mathsf{lookup\_sessions}(\sigma')(s_{id})) = s \\ \mathsf{update\_sessions}(\sigma', s_{id}, s) = \sigma''\end{array}}{\Gamma, \phi, \sigma, \rho \models_{\mathrm{e}} \texttt{accept}\ \ \texttt{(}exp\texttt{)} : \sigma'', s}$$

$$\frac{\begin{array}{c}\Gamma, \phi, \sigma, \rho \models_{\mathrm{e}} exp : \sigma', s_{id} \\ \mathsf{acccept}(s_{id}) = \bot \\ \mathsf{update\_sessions}(\sigma', s_{id}, \bot) = \sigma''\end{array}}{\Gamma, \phi, \sigma, \rho \models_{\mathrm{e}} \texttt{accept}\ \ \texttt{(}exp\texttt{)} : \sigma'', \bot} \qquad \frac{\Gamma, \phi, \sigma, \rho \models_{\mathrm{e}} exp : \sigma', \bot}{\Gamma, \phi, \sigma, \rho \models_{\mathrm{e}} \texttt{accept}\ \ \texttt{(}exp\texttt{)} : \sigma', \bot}$$

Bind

$$\frac{\begin{array}{c}\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \sigma_1, v \qquad v \neq \bot \\ \Gamma,\phi,\sigma_1,\rho \models_{\text{e}} exp_1 : \sigma_2, v_1 \qquad v_1 \neq \bot \\ \Gamma,\phi,\sigma_2,\rho \models_{\text{e}} exp_2 : \sigma_3, v_2 \qquad v_2 \neq \bot \\ \mathsf{update\_bridges}(\sigma_3, v_1, \langle v, v_1, v_2\rangle) = \sigma_4 \\ \mathsf{update\_bridges}(\sigma_4, v_2, \langle v, v_1, v_2\rangle) = \sigma_5 \\ \mathsf{BridgeInst}(v, v_1, v_2) = b\end{array}}{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp\texttt{.bind (}exp_1\texttt{, }exp_2\texttt{)} : \sigma_5, b}$$

$$\frac{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \sigma_1, \bot}{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp\texttt{.bind (}exp_1\texttt{, }exp_2\texttt{)} : \sigma_1, \bot}$$

$$\frac{\begin{array}{c}\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \sigma_1, v \qquad v \neq \bot \\ \Gamma,\phi,\sigma_1,\rho \models_{\text{e}} exp_1 : \sigma_2, \bot\end{array}}{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp\texttt{.bind (}exp_1\texttt{, }exp_2\texttt{)} : \sigma_2, \bot}$$

$$\frac{\begin{array}{c}\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \sigma_1, v \qquad v \neq \bot \\ \Gamma,\phi,\sigma_1,\rho \models_{\text{e}} exp_1 : \sigma_2, v_1 \qquad v_1 \neq \bot \\ \Gamma,\phi,\sigma_2,\rho \models_{\text{e}} exp_2 : \sigma_3, \bot\end{array}}{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp\texttt{.bind (}exp_1\texttt{, }exp_2\texttt{)} : \sigma_3, \bot}$$

Command

$$\frac{\begin{array}{c}\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \mathsf{ServiceInst}(1, 1, \langle srv_1; \ldots; srv_p\rangle) \\ \Gamma,\phi,\sigma,\rho \models_{\text{e}} exp_1 : \sigma_1, v_1 \\ \ldots \\ \Gamma,\phi,\sigma_{n-1},\rho \models_{\text{e}} exp_n : \sigma_n, v_n \\ \forall i \in [1; \ldots; n], v_i \neq \bot \\ \mathsf{message}(cmdName, \langle v_1; \ldots; v_n\rangle, 1, 1, \langle srv_1; \ldots; srv_p\rangle) = \mathsf{true}, \langle w_1\rangle\end{array}}{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp\texttt{.}cmdName\texttt{ (}exp_1\texttt{, ... , }exp_n\texttt{)} : \sigma_n, w_1}$$

$$\frac{\begin{array}{c}\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \mathsf{ServiceInst}(min, rq, \langle srv_1; \ldots; srv_p\rangle) \\ j = rq \triangledown p \\ \Gamma,\phi,\sigma,\rho \models_{\text{e}} exp_1 : \sigma_1, v_1 \\ \ldots \\ \Gamma,\phi,\sigma_{n-1},\rho \models_{\text{e}} exp_n : \sigma_n, v_n \\ \forall i \in [1; \ldots; n], v_i \neq \bot \\ \mathsf{message}(cmdName, \langle v_1; \ldots; v_n\rangle, min, j, \langle srv_1; \ldots; srv_p\rangle) = \mathsf{true}, \langle w_1; \ldots; w_j\rangle\end{array}}{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp\texttt{.}cmdName\texttt{ (}exp_1\texttt{, ... , }exp_n\texttt{)} : \sigma_n, \langle w_1; \ldots; w_j\rangle}$$

$$\frac{\begin{array}{c}\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \mathsf{ServiceInst}(min, rq, \langle srv_1; \ldots; srv_p\rangle) \\ j = rq \triangledown p \\ \Gamma,\phi,\sigma,\rho \models_{\text{e}} exp_1 : \sigma_1, v_1 \\ \ldots \\ \Gamma,\phi,\sigma_{n-1},\rho \models_{\text{e}} exp_n : \sigma_n, v_n \\ \exists i \in [1;\ldots;n], v_i = \bot \vee \mathsf{message}(cmdName, \langle v_1;\ldots;v_n\rangle, min, j, \langle srv_1;\ldots;srv_p\rangle) = \mathsf{false}, \_\end{array}}{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp\,\texttt{.}\,cmdName\,\texttt{(}\,exp_1\texttt{,}\ \ldots\ \texttt{,}\ exp_n\texttt{)} : \sigma_n, \bot}$$

$$\frac{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \sigma', \bot}{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp\,\texttt{.}\,cmdName\,\texttt{(}\,exp_1\texttt{,}\ \ldots\ \texttt{,}\ exp_n\texttt{)} : \sigma', \bot}$$

Field

$$\frac{\begin{array}{c}\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \mathsf{ServiceInst}(1, 1, \langle srv_1; \ldots; srv_p\rangle \\ \mathsf{message}(field, \langle\rangle, 1, 1, \langle srv_1;\ldots;srv_p\rangle) = \mathsf{true}, \langle v_1\rangle\end{array}}{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp\,\texttt{.}\,field : \sigma, v_1}$$

$$\frac{\begin{array}{c}\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \mathsf{ServiceInst}(i, j, \langle srv_1; \ldots; srv_p\rangle) \\ k = j \triangledown p \\ \mathsf{message}(field, \langle\rangle, i, k, \langle srv_1;\ldots;srv_p\rangle) = \mathsf{true}, \langle v_1;\ldots;v_n\rangle\end{array}}{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp\,\texttt{.}\,field : \sigma, \langle v_1;\ldots;v_n\rangle}$$

$$\frac{\begin{array}{c}\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \mathsf{ServiceInst}(i, j, \langle srv_1; \ldots; srv_p\rangle) \\ k = j \triangledown p \\ \mathsf{message}(field, \langle\rangle, i, k, \langle srv_1;\ldots;srv_p\rangle) = \mathsf{false}, \_\end{array}}{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp\,\texttt{.}\,field : \sigma, \bot}$$

$$\frac{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \sigma', \bot}{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp\,\texttt{.}\,field : \sigma', \bot}$$

D.5.5 Semantics of Declarations

Variable

$$\frac{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \sigma', v \qquad v \neq \bot}{\Gamma,\phi,\sigma,\rho \models_{\text{s}} \texttt{var}\ \ type\ id\ \texttt{=}\ exp\,\texttt{;} : \sigma', \rho[id \mapsto v], \top}$$

$$\frac{\Gamma,\phi,\sigma,\rho \models_{\text{e}} exp : \sigma', \bot}{\Gamma,\phi,\sigma,\rho \models_{\text{s}} \texttt{var}\ \ type\ id\ \texttt{=}\ exp\,\texttt{;} : \sigma', \rho[id \mapsto \bot], \bot}$$

Service

$$\frac{\begin{array}{c}\Gamma \models_{\text{ph}} \mathit{servicePath} : \mathsf{Service}(\mathit{srvLabel}, \_, \_, \mathit{props}) \\ \Gamma, \phi, \sigma, \rho, \mathit{props} \models_{\text{ps}} \mathit{properties} : \sigma', \mathit{props}' \\ \mathsf{Service}(\mathit{srvLabel}, 1, 1, \mathit{props}') = v\end{array}}{\Gamma, \phi, \sigma, \rho \models_{\text{s}} \texttt{service<}\mathit{servicePath}\texttt{>}\ \ \{\mathit{properties}\}\, id : \sigma', \rho[id \mapsto v], \top}$$

Event

$$\frac{\begin{array}{c}\Gamma \models_{\text{ph}} \mathit{userTypePath} : \mathsf{User}(\mathit{label}, \mathit{props}) \\ \Gamma, \phi, \sigma, \rho, \mathit{props} \models_{\text{ps}} \mathit{properties} : \sigma', \mathit{props}' \\ \mathsf{Event}(\mathit{label}, \mathit{props}') = e\end{array}}{\Gamma, \phi, \sigma, \rho \models_{\text{s}} \texttt{event<}\mathit{userTypePath}\texttt{>}\ \ \{\mathit{properties}\}\, id : \sigma', \rho[id \mapsto e], \top}$$

$$\frac{\begin{array}{c}\Gamma \models_{\text{ph}} \mathit{userTypePath} : \mathsf{User}(\mathit{label}, \mathit{props}) \\ \Gamma, \phi, \sigma, \rho, \mathit{props} \models_{\text{ps}} \mathit{properties} : \sigma', \mathit{props}' \\ \Gamma, \phi, \sigma, \rho \models_{\text{e}} \mathit{service} : \mathsf{Service}(id_{srv}, \_, \_, \mathit{prop}_{srv}) \\ \mathsf{Service}(id_{srv}, i, i, \mathit{prop}_{srv}) = \mathit{srv} \\ \mathsf{RequiredEvent}(\mathit{label}, \mathit{srv}, \mathit{props}') = e\end{array}}{\Gamma, \phi, \sigma, \rho \models_{\text{s}} \texttt{event<}\mathit{userTypePath}\texttt{>}\ \ \texttt{from}\ \ \mathit{service}\texttt{[}i\texttt{]}\ \ \{\mathit{properties}\}\, id : \sigma', \rho[id \mapsto e], \top}$$

$$\frac{\begin{array}{c}\Gamma \models_{\text{ph}} \mathit{userTypePath} : \mathsf{User}(\mathit{label}, \mathit{props}) \\ \Gamma, \phi, \sigma, \rho, \mathit{props} \models_{\text{ps}} \mathit{properties} : \sigma', \mathit{props}' \\ \Gamma, \phi, \sigma, \rho \models_{\text{e}} \mathit{service} : \mathsf{Service}(id_{srv}, \_, \_, \mathit{prop}_{srv}) \\ \mathsf{Service}(id_{srv}, 1, \top, \mathit{prop}_{srv}) = \mathit{srv} \\ \mathsf{RequiredEvent}(\mathit{label}, \mathit{srv}, \mathit{props}') = e\end{array}}{\Gamma, \phi, \sigma, \rho \models_{\text{s}} \texttt{event<}\mathit{userTypePath}\texttt{>}\ \ \texttt{from}\ \ \mathit{service}\texttt{[}\star\texttt{]}\ \ \{\mathit{properties}\}\, id : \sigma', \rho[id \mapsto e], \top}$$

Session

$$\frac{\begin{array}{c}\Gamma \models_{\text{ph}} \mathit{userTypePath} : \mathsf{User}(\mathit{label}, \mathit{props}) \\ \Gamma, \phi, \sigma, \rho, \mathit{props} \models_{\text{ps}} \mathit{properties} : \sigma', \mathit{props}' \\ \mathsf{lookup\_self}(\sigma') = (\mathit{srvName}, \_) \\ \mathsf{Session}(\mathit{srvName}, \top, \mathsf{pending}, \mathit{props}') = s\end{array}}{\Gamma, \phi, \sigma, \rho \models_{\text{s}} \texttt{session<}\mathit{userTypePath}\texttt{>}\ \ \{\mathit{properties}\}\, id : \sigma', \rho[id \mapsto s], \top}$$

$$\frac{\begin{array}{c}\Gamma \models_{\text{ph}} \mathit{userTypePath} : \mathsf{User}(\mathit{label}, \mathit{props}) \\ \Gamma, \phi, \sigma, \rho, \mathit{props} \models_{\text{ps}} \mathit{properties} : \sigma', \mathit{props}' \\ \Gamma, \phi, \sigma', \rho \models_{\text{e}} \mathit{service} : \mathit{srv} \\ \mathsf{ProvidedSession}(\mathit{label}, \mathit{srv}, \mathit{props}') = s\end{array}}{\Gamma, \phi, \sigma, \rho \models_{\text{s}} \texttt{session<}\mathit{userTypePath}\texttt{>}\ \ \texttt{from}\ \ \mathit{service}\texttt{[}\star\texttt{]}\ \ \{\mathit{properties}\}\, id : \sigma', \rho[id \mapsto s], \top}$$

onReceive

$$\frac{\begin{array}{c} cmpd = \{\, stmts \,\} \\ \mathsf{Reception}(imType, im, stmts) = r \\ \mathsf{update\_env}(\sigma, id, r) = \sigma' \end{array}}{\Gamma, \phi, \sigma, \rho \models_{\mathrm{s}} \texttt{onReceive}\ \ id\ \ \texttt{(}imType\ \ im\texttt{)}\ \ \ cmpd : \sigma', \rho, \top}$$

Bridge

$$\frac{\begin{array}{c} cmpd = \{\, stmts \,\} \\ \mathsf{Bridge}(type, stmts) = b \\ \mathsf{update\_env}(\sigma, id, b) = \sigma' \end{array}}{\Gamma, \phi, \sigma, \rho \models_{\mathrm{s}} \texttt{bridge<}type\texttt{>}\ \ \ cmpd\ \ \ id : \sigma', \rho, \top}$$

D.5.6 Rules Relative to the Domain

$$\frac{\Sigma(\langle Service \rangle) = \mathbf{Service}(\langle Service \rangle, \mu, prop)}{(\Sigma, \Xi, \Delta) \models_{\mathrm{ph}} \texttt{Service} : \mathbf{Service}(\langle Service \rangle, \mu, prop)}$$

$$\frac{\begin{array}{c} (\Sigma, \Xi, \Delta) \models_{\mathrm{ph}} pServicePath : \mathbf{Service}(pSrvLabel, \mu_1, prop_1) \\ \Sigma(pSrvLabel +\!\!+ \langle node \rangle) = \mathbf{Service}(srvLabel, \mu_1, prop_2) \\ \mu = \mu_1 +\!\!+ \mu_2 \qquad prop = prop_1 +\!\!+ prop_2 \end{array}}{(\Sigma, \Xi, \Delta) \models_{\mathrm{ph}} pServicePath.node : \mathbf{Service}(srvLabel, \mu, prop)}$$

$$\frac{\Delta(\langle node \rangle) = \mathbf{UserType}(\langle node \rangle, prop)}{(\Sigma, \Xi, \Delta) \models_{\mathrm{ph}} node : \mathbf{UserType}(\langle node \rangle, prop)}$$

$$\frac{\begin{array}{c} (\Sigma, \Xi, \Delta) \models_{\mathrm{ph}} pUserTypePath : \mathbf{UserType}(pUTLabel, prop_1) \\ \Delta(pUTLabel +\!\!+ \langle node \rangle) = \mathbf{UserType}(utLabel, prop_2) \\ prop = prop_1 +\!\!+ prop_2 \end{array}}{(\Sigma, \Xi, \Delta) \models_{\mathrm{ph}} pUserTypePath\,\texttt{.}\,node : \mathbf{UserType}(utLabel, prop)}$$

$$\frac{\begin{array}{c} \Gamma, \phi, \sigma, \rho, prop \models_{\mathrm{p}} property_1 : \sigma_1, prop_1 \\ \cdots \\ \Gamma, \phi, \sigma_{n-1}, \rho, prop_{n-1} \models_{\mathrm{p}} property_n : \sigma_n, prop_n \end{array}}{\Gamma, \phi, \sigma, \rho, prop \models_{\mathrm{ps}} property_1 \ldots property_n : \sigma_n, prop_n}$$

$$\frac{\Gamma, \phi, \sigma, \rho \models_{\mathrm{e}} exp : \sigma', v}{\Gamma, \phi, \sigma, \rho, prop \models_{\mathrm{s}} id\ \ =\ \ exp\ \texttt{;}\ : \sigma', prop[id \mapsto v]}$$

$$\frac{\Gamma, \models_{\text{ph}} \mathit{UserTypePath} : \mathbf{UserType}(\mathit{utLabel}, \mathit{utProp}) \quad \Gamma, \phi, \sigma, \rho, \mathit{utProp} \models_{\text{ps}} \mathit{properties} : \sigma', v}{\Gamma, \phi, \sigma, \rho, \mathit{prop} \models_{\text{p}} \mathit{property} \ = \ \mathit{UserTypePath} \ \ \{ \mathit{properties} \ \ \} : \sigma', \mathit{prop}[\mathit{property} \mapsto v]}$$

D.6 SIP Virtual Machine API

The following functions define the available functions that may be called by the interpreter or generated code.

- REGISTER : Id × bool → response$_{SIP}$
- INVITE : Id$_{SRV}$ × properties → response$_{SIP}$
- INVITE_try : Id$_S$ → unit
- INVITE_ok : Id$_S$ → unit
- INVITE_decline : Id$_S$ → unit
- BYE : Id$_S$ → unit
- BYE_ok : Id$_S$ → unit
- NOTIFY_try : Id$_E$ → unit
- NOTIFY_ok : Id$_E$ → unit
- NOTIFY_error : Id$_E$ → unit
- MESSAGE : Id$_{SRV}$ × cmd/field × value list → response$_{SIP}$
- MESSAGE_try : Id$_M$ → unit
- MESSAGE_ok : Id$_M$ × value → unit
- MESSAGE_error : Id$_M$ → unit
- SUBSCRIBE : Id$_{SRV}$ × Id$_D$ × bool → response$_{SIP}$
- PUBLISH : EventMsg → response$_{SIP}$

The virtual machine used by Pantaxou can then be defined in term of the lower level operations provided by the SIP virtual machine.

register and unregister Functions

The first REGISTER argument is the SIP URI of the current service, i.e., its name. The second REGISTER argument indicates if the expire header is set to a strictly positive constant value (true) or to zero (false). The argument thus allows to register and un-register on the SIP server.

$$\frac{\mathsf{REGISTER}(\mathit{name}, \mathsf{true}) = \mathsf{OK}(\_)}{\mathsf{register}(\langle \mathit{name}, \mathit{srv} \rangle) = \mathsf{true}} \qquad \frac{\mathsf{REGISTER}(\mathit{name}, \mathsf{true}) = \mathsf{ERROR}}{\mathsf{register}(\langle \mathit{name}, \mathit{srv} \rangle) = \mathsf{false}}$$

$$\frac{\mathsf{REGISTER}(\mathit{name}, \mathsf{false}) = \_}{\mathsf{unregister}(\langle \mathit{name}, \mathit{srv} \rangle) = ()}$$

discover Function

The discover function relies on a framework service with a special name, framework. This service provides the discover operation.

$$\frac{\mathsf{MESSAGE}(\mathsf{framework}, \mathsf{discover}, \langle \mathit{srv}, \mathit{properties} \rangle) = \mathsf{OK}(\mathit{services})}{\mathsf{discover}(\mathit{srv}, \mathit{properties}) = \mathit{services}}$$

$$\frac{\mathsf{MESSAGE}(\mathsf{framework}, \mathsf{discover}, \langle srv, properties\rangle) = \mathsf{ERROR}}{\mathsf{discover}(srv, properties) = \langle\rangle}$$

message **Function**

$$\frac{\begin{array}{c} rq > 0 \\ \mathsf{MESSAGE}(srv_{id_n}, op, params) = \mathsf{OK}(v_n) \\ \mathsf{message}(op, params, min-1, rq-1, \langle srv_{id_1}; \ldots; srv_{id_{n-1}}\rangle) = sol, solList \end{array}}{\mathsf{message}(op, params, min, rq, \langle srv_{id_1}; \ldots; srv_{id_n}\rangle) = sol, solList +\!\!+ \langle v_n\rangle}$$

$$\frac{\begin{array}{c} rq > 0 \\ \mathsf{MESSAGE}(srv_{id_n}, op, params) = \mathsf{ERROR} \\ n - 1 \geq min \\ \mathsf{message}(op, params, min, rq, \langle srv_{id_1}; \ldots; srv_{id_{n-1}}\rangle) = sol, solList \end{array}}{\mathsf{message}(op, params, min, rq, \langle srv_{id_1}; \ldots; srv_{id_n}\rangle) = sol, solList}$$

$$\frac{\begin{array}{c} rq > 0 \\ \mathsf{MESSAGE}(id, srv_{id_n}, label) = \mathsf{ERROR} \\ n - 1 < min \end{array}}{\mathsf{message}(op, params, min, rq, \langle srv_{id_1}; \ldots; srv_{id_n}\rangle) = \mathsf{false}, \langle\rangle}$$

$$\frac{rq = 0}{\mathsf{message}(op, params, min, rq, \langle \ldots\rangle) = \mathsf{true}, \langle\rangle}$$

subscribe **Function**

$$\frac{\begin{array}{c} rq > 0 \\ \mathsf{SUBSCRIBE}(srv_{id_n}, label, \mathsf{true}) = \mathsf{OK} \\ \mathsf{subscribe}(label, min-1, rq-1, \langle srv_{id_1}; \ldots; srv_{id_{n-1}}\rangle) = sol, srvList \end{array}}{\mathsf{subscribe}(label, min, rq, \langle srv_{id_1}; \ldots; srv_{id_n}\rangle) = sol, srvList +\!\!+ \langle srv_{id_n}\rangle}$$

$$\frac{\begin{array}{c} rq > 0 \\ \mathsf{SUBSCRIBE}(srv_{id_n}, label, \mathsf{true}) = \mathsf{ERROR} \\ n - 1 \geq min \\ \mathsf{subscribe}(label, min, rq, \langle srv_{id_1}; \ldots; srv_{id_{n-1}}\rangle) = sol, srvList \end{array}}{\mathsf{subscribe}(label, min, rq, \langle srv_{id_1}; \ldots; srv_{id_n}\rangle) = sol, srvList}$$

$$\frac{\begin{array}{c} rq > 0 \\ \mathsf{SUBSCRIBE}(srv_{id_n}, label, \mathsf{true}) = \mathsf{ERROR} \\ n - 1 < min \end{array}}{\mathsf{subscribe}(label, min, rq, \langle srv_{id_1}; \ldots; srv_{id_n}\rangle) = \mathsf{false}, \langle\rangle}$$

$$\frac{rq = 0}{\mathsf{subscribe}(label, min, rq, \langle \ldots\rangle) = \mathsf{true}, \langle\rangle}$$

unsubscribe **Function**

$$\frac{\begin{array}{c} \mathsf{SUBSCRIBE}(srv_{id_n}, label, \mathsf{false}) \\ \mathsf{unsubscribe}(label, \langle srv_{id_1}; \ldots; srv_{id_{n-1}}\rangle) \end{array}}{\mathsf{unsubscribe}(label, \langle srv_{id_1}; \ldots; srv_{id_n}\rangle) = ()}$$

$$\frac{}{\mathsf{unsubscribe}(label, \langle\rangle) = ()}$$

publish **Function**

$$\frac{\mathsf{lookup\_self}(\sigma) = (srvName, \_) \quad \mathsf{PUBLISH}(\mathsf{EventMsg}(srvName, v)) = \mathsf{OK}}{\mathsf{publish}(\sigma, v) : \top}$$

$$\frac{\mathsf{lookup\_self}(\sigma) = (srvName, \_) \quad \mathsf{PUBLISH}(\mathsf{EventMsg}(srvName, v)) = \mathsf{ERROR}}{\mathsf{publish}(\sigma, v) : \bot}$$

invite **Function**

$$\frac{\begin{array}{c} rq > 0 \\ v = \mathsf{RequiredSession}(sesslabel, sessProperties) \\ \mathsf{INVITE}(srv_n, sessProperties) = \mathsf{OK}(s_n) \\ \mathsf{update\_sessions}(\sigma, sesslabel, \mathsf{Session}(srv_n, s_n, \mathsf{pending}, sessProperties)) = \sigma' \\ \mathsf{invite}(\sigma', v, min - 1, rq - 1, \langle srv_1; \ldots; srv_{n-1} \rangle) = \sigma'', sol, solList \end{array}}{\mathsf{invite}(\sigma, v, min, rq, \langle srv_1; \ldots; srv_n \rangle) = \sigma'', sol, solList +\!\!+ \langle s_n \rangle}$$

$$\frac{\begin{array}{c} rq > 0 \\ v = \mathsf{RequiredSession}(sesslabel, sessProperties) \\ \mathsf{INVITE}(srv_n, sessProperties) = \mathsf{ERROR} \\ n - 1 \geq min \\ \mathsf{invite}(\sigma, v, min, rq, \langle srv_1; \ldots; srv_{n-1} \rangle) = \sigma', sol, solList \end{array}}{\mathsf{invite}(\sigma, v, min, rq, \langle srv_1; \ldots; srv_n \rangle) = \sigma', sol, solList}$$

$$\frac{\begin{array}{c} rq > 0 \\ v = \mathsf{RequiredSession}(sesslabel, sessProperties) \\ \mathsf{INVITE}(srv_n, sessProperties) = \mathsf{ERROR} \\ n - 1 < min \end{array}}{\mathsf{invite}(\sigma, v, min, rq, \langle srv_1; \ldots; srv_n \rangle) = \sigma, \mathsf{false}, \langle\rangle}$$

$$\frac{rq = 0}{\mathsf{invite}(\sigma, v, min, rq, \langle \ldots \rangle) = \sigma, \mathsf{true}, \langle\rangle}$$

bye, decline **and** accept **Functions**

$$\frac{\mathsf{BYE}(s_{id})}{\mathsf{bye}(s_{id}) = ()} \qquad \frac{\mathsf{INVITE\_decline}(s_{id})}{\mathsf{decline}(s_{id}) = ()}$$

$$\frac{\mathsf{INVITE\_ok}(s_{id}) \Rightarrow \mathsf{ACK}}{\mathsf{accept}(s_{id}) = \top} \qquad \frac{\mathsf{INVITE\_ok}(s_{id}) \not\Rightarrow \mathsf{ACK}}{\mathsf{accept}(s_{id}) = \bot}$$

Response Functions

The definition of the following Pantaxou virtual machine functions are straightforward in the SIP virtual machine. They just call the corresponding SIP virtual machine functions with the same arguments. $\mathsf{message}_{try}$ is given as an example.

- $\mathsf{message}_{try/error} : \mathsf{id}_{message} \rightarrow \mathsf{unit}$
- $\mathsf{message}_{ok} : \mathsf{id}_{message} \times \mathsf{value} \rightarrow \mathsf{unit}$
- $\mathsf{notify}_{try/error/ok} : \mathsf{id}_{event} \rightarrow \mathsf{unit}$
- $\mathsf{invite}_{try} : \mathsf{id}_{session} \rightarrow \mathsf{unit}$
- $\mathsf{bye}_{ok} : \mathsf{id}_{session} \rightarrow \mathsf{unit}$

$$\frac{\mathsf{MESSAGE\_try}(id_m)}{\mathsf{message}_{try}(id_m) = ()}$$

Annexe E

Gestionnaire de lumière

```
1  package fr.labri.phoenix.amigo.lightmanager.impl;
2
3  import java.util.LinkedList;
4
5  import fr.labri.phoenix.amigo.event.LuminosityEvent;
6  import fr.labri.phoenix.amigo.functionality.LuminosityEventInput;
7  import fr.labri.phoenix.amigo.LightManager;
8
9  import fr.labri.phoenix.amigo.partition.LightManagerPart;
10 import fr.labri.phoenix.amigo.intf.ILightManager;
11 import fr.labri.phoenix.amigo.Light;
12
13 import fr.labri.phoenix.amigo.partition.LightPart;
14 import fr.labri.phoenix.amigo.intf.ILight;
15 import fr.labri.phoenix.amigo.LightSensor;
16
17 import fr.labri.phoenix.amigo.partition.LightSensorPart;
18 import fr.labri.phoenix.amigo.intf.ILightSensor;
19 import fr.labri.phoenix.amigo.pantaxou.PantaxouComponent;
20
21 public class LightManagerImpl extends LightManager implements Runnable {
22
23   private LinkedList<ILight> getLights() {
24     LightPart lightPart = Light.getPartition(this);
25     return Light.getServices(lightPart);
26   }
27
28   private LinkedList<ILight> getLight(int n) {
29     LinkedList<ILight> light = getLights();
30     int size = light.size();
31     int j = size - Math.min(n,size);
32     for (int i = 0; i < j; i++) {
33       light.removeLast();
34     }
35     return light;
36   }
37
38   private LinkedList<ILightSensor> getLightSensors() {
39     LightSensorPart lightSensorPart = LightSensor.getPartition(this);
40     return LightSensor.getServices(lightSensorPart);
41   }
42
43   private LinkedList<ILightSensor> getLightSensor(int n) {
44     LinkedList<ILightSensor> lightSensor = getLightSensors();
45     int size = lightSensor.size();
46     int j = size - Math.min(n,size);
47     for (int i = 0; i < j; i++) {
48       lightSensor.removeLast();
```

```
49      }
50      return lightSensor;
51    }
52
53    protected abstract class LumEvt {
54      public abstract void receive(LuminosityEvent e);
55    }
56
57    private LumEvt lumEvt = null;
58
59    public void receive(LuminosityEvent e) {
60      if (lumEvt != null) {
61        lumEvt.receive(e);
62      }
63    }
64
65    private class LumEvtReception extends LumEvt {
66      public void receive(LuminosityEvent e) {
67        int lum = e.getData().getLuminosityVal();
68        if (lum < 30) {
69          LinkedList<ILight> lights = getLights();
70          for(ILight light: lights)
71            light.on();
72        } else {
73          if (lum > 40) {
74            LinkedList<ILight> lights = getLights();
75            for(ILight light: lights)
76              light.off();
77          }
78        }
79      }
80    }
81
82    public LightManagerImpl(PantaxouComponent component) {
83      super(component);
84    }
85
86    public void run() {
87      lumEvt = new LumEvtReception();
88      LinkedList<ILightSensor> lightSensors = getLightSensors();
89      for (ILightSensor lightSensor: lightSensors) {
90        lightSensor.subscribe(this);
91      }
92    }
93  }
```