Approche langage au développement du support protocolaire d'applications réseaux

Laurent Burgy

N° d'ordre: 3585

THÈSE

présentée à

L'UNIVERSITÉ BORDEAUX 1
École Doctorale de Mathématiques et Informatique

par **Laurent BURGY**

pour obtenir le grade de :

DOCTEUR
Spécialité: INFORMATIQUE

Sujet :

Approche langage au développement du support protocolaire d'applications réseaux

Soutenue le : 28 Avril 2008

Après avis de :

MM. :	Jean-Charles	FABRE,	Professeur, INP-ENSEEIHT	Rapporteurs
	Pierre	PARADINAS,	Professeur, INRIA	

Devant la commission d'examen formée de :

M. :	Jean	ROMAN	Professeur, ENSEIRB	Président
MM. :	Jean-Charles	FABRE,	Professeur, INP-ENSEEIHT	Rapporteurs
	Pierre	PARADINAS,	Professeur, INRIA	
MM. :	Yolande	BERBERS,	Professeur, KULeuven	Examinateurs
	Gilles	MULLER,	Professeur, École des Mines de Nantes	
	Charles	CONSEL,	Professeur, ENSEIRB	
	Laurent	RÉVEILLÈRE,	Maître de Conférences, ENSEIRB	

à Katia,

Remerciements

Lors de l'écriture de son document, quand on ne peut dormir la nuit, on s'imagine souvent en train de faire ce que je fais. Bien souvent, on suppose que ce moment sera plaisant. Finalement, écrire des remerciements s'avère au moins aussi délicat que n'importe quelle autre partie du document.

Tout d'abord, remercions les membres du jury :

– Jean Roman, professeur des Universités à l'ENSEIRB, responsable de l'équipe Scalapplix, qui a présidé ce jury.
– Jean-Charles Fabre, professeur des Universités à l'Institut National Polytechnique de Toulouse, et Pierre Paradinas de l'INRIA et du CNAM qui ont accepté la charge de rapporteur du document. Leurs avis et commentaires m'ont permis d'améliorer encore cette thèse. Les différentes discussions que nous avons pues avoir furent vraiment constructives.
– Yolande Berbers, professeur à la Katholieke Universiteit Leuven et Gilles Muller, professeur à l'École de Mines de Nantes, d'avoir lu en détails le document et d'avoir bravé les affres des transports pour assister à ma soutenance.

Je souhaite remercier mon directeur de thèse, Charles Consel, responsable de l'équipe Phoenix de m'avoir permis d'effectuer ce travail en m'accueillant au sein de son équipe. Au-delà du travail accompli, je retiendrai les discussions que nous avons pues avoir auprès de la machine à café concernant la recherche et le monde (plat) en général.

Laurent Réveillère mériterait une page entière de remerciements, et encore, une page ne suffirait certainement pas. Laurent a été d'un grand soutien tant moral qu'au niveau du travail et des directions à donner à ma thèse en se montrant très pédagogue. Il a supporté mes humeurs changeantes et il a su me secouer aux bons moments.

Gilles Muller et Julia Lawall ont été des éléments déterminants dans l'accomplissement de ces travaux. Leur support et leurs critiques m'ont permis de faire progresser Zebu et de me faire progresser au niveau de la rigueur. Julia a également risqué sa vie en traversant les rues de Beijing avec moi.

J'ai pu profiter de l'aura de Sapan Bhatia dans ses derniers mois au sein de l'équipe Phoenix.

Dans ces derniers mois, David Bromberg incarnait le rôle du doctorant devenu docteur récemment. Ses conseils, ses remarques m'ont beaucoup servi pour sortir de ma chrysalide et devenir un papillon.

Nicolas Palix a toujours été là et ça n'a pas dû être forcément évident de partager un bureau avec moi. Il m'a souvent écouté, conseillé et aidé. Ces compétences techniques ont été un plus indéniable à mon travail.

Wilfried *Wiwi* Jouve a partagé aussi mon bureau, en a beaucoup souffert et doit en garder quelques stigmates ou terreurs noctures. Hormis des goûts musicaux un peu *chelous* voire déplorables, Wiwi m'a permis de bien rigoler (souvent à ces dépens).

Je remercie mes estimés collègues, Julien Lancia, Julien Mercadal et Zoé *Dr.* Drey. Je pourrais faire de longues phrases avec des mots compliqués qui s'en trouveraient galvaudés mais je ne souhaite pas sombrer dès potron-minet dans un style trop ostentatoire.

Merci également à mes partenaires de baby, Benjamin et Julien *le stagiaire*, d'avoir fait semblant d'être aussi mauvais pour me laisser gagner.

Merci à Sabine Baussart.

Merci à Patrice Kadionik pour l'aide concernant les systèmes embarqués.

Merci à Renaud Marlet pour ses encouragements.

Après ces remerciements professionnels d'usage, passons à l'aspect personnel :

Je remercie Cédric et Loïc pour tout ce qu'on a fait et d'avoir refait le monde avec moi.

Je remercie mes amis, Fabrice, Florent, Julien, Bertrand, Maxime, Thomas, Cédric D., Cédric B., Rémy et Vanessa et Fabien.

Je remercie Antoine pour le voisinage durant l'ENSEIRB et la *hotline* ultra-efficace.

Je remercie affectueusement toute ma famille. Mes parents, Chantal et Christian, pour m'avoir fait aussi intelligent et modeste et d'avoir financé ma scolarité qui s'est avérée bien longue au final. Je remercie mes soeurs, Valérie et Sandrine, et beaux-frères, Thierry et Patrice, mais je leur en veux toujours un peu d'avoir triché au Cluedo. Je remercie mon neveu et mes nièces, Ophélie, Anthony, Tiphanie et Emma parce qu'ils se moquent de savoir si j'ai un papier d'accepté ou pas. Je remercie mes beaux-parents, Dany et Domi d'avoir conçu ma femme (et son frère dans une moindre mesure) et de m'avoir bien nourri.

Je remercie enfin ma femme, Katia, pour tout ce qu'elle a enduré et son soutien indéfectible, m'encourageant toujours, dans un total désintérêt, à trouver ce qui est le mieux pour moi. J'espère qu'elle obtiendra l'Ordre National du Mérite.

Résumé

Une application réseau communique avec d'autres applications par le biais d'un ensemble consensuel de règles régissant la communication, appelé protocole. Cette communication est gérée par la partie de l'application connue comme la couche de support protocolaire qui gère la manipulation de messages protocolaires. Elle s'avère être un composant critique d'une application réseau puisqu'elle représente l'interface entre celle-ci et le monde extérieur. Elle est donc soumise à deux contraintes fortes : une contrainte d'efficacité pour pouvoir traiter un grand nombre de messages et une contrainte de robustesse pour faire face à des attaques visant à déstabiliser l'application ou la plate-forme matérielle sous-jacente. Malgré ces contraintes, le processus de développement de cette couche demeure rudimentaire et requiert un haut niveau d'expertise. Il consiste à traduire manuellement une spécification du protocole écrite dans un formalisme haut niveau comme ABNF vers du code bas niveau tel que du C. Le fossé entre ces niveaux d'abstraction favorise l'apparition d'erreurs.

Cette thèse propose une approche langage au développement de la couche de support protocolaire d'applications réseaux, pour améliorer leur robustesse sans compromettre leur performance. Notre approche est fondée sur l'utilisation d'un langage dédié, Zebu, pour décrire la spécification des couches de support protocolaire d'applications réseaux qui utilisent des protocoles applicatifs textuels *à la* HTTP. La syntaxe de Zebu est très proche de celles du formalisme ABNF, favorisant ainsi l'adoption de Zebu par des experts du domaine. En annotant la spécification ABNF d'un protocole, l'utilisateur de Zebu peut adapter une couche de support protocolaire à une application donnée. Dans un premier temps, le compilateur Zebu vérifie la spécification annotée pour déceler d'éventuelles incohérences. Ensuite, une couche de support protocolaire définie par les annotations fournies est générée automatiquement. Cette couche consiste en un ensemble de structures de données pour représenter un message, un analyseur syntaxique qui remplit ces structures de données et des fonctions utilitaires pour l'accès à ces données ou piloter l'analyse syntaxique des messages. Par défaut, l'analyseur syntaxique de messages généré n'accepte que les messages respectant scrupuleusement la spécification. Ce critère de validation peut être modifié pour plus de flexibilité ou de meilleures performances.

Les contributions de cette thèse sont les suivantes :

- Nous avons effectué une analyse complète des applications réseaux. Grâce à cette analyse, nous avons identifié les paramètres d'entrée à la conception d'un langage dédié.
- Nous avons conçu un langage dédié déclaratif, Zebu, permettant de spécifier la couche de support protocolaire propre à une application. Diverses spécifications ont été développées afin de valider le pouvoir d'expression du langage.
- Nous avons développé un compilateur Zebu générant du code C pour le système d'exploitation Linux et une chaîne de compilation optimisée ciblant les systèmes embarqués. Nous avons effectué des expérimentations afin de mesurer l'impact de notre approche sur la robustesse des couches de support protocolaire. Cette étude nous a permis de montrer qu'une couche de support protocolaire fondée sur Zebu détecte au minimum quatre fois plus de messages invalides qu'une couche de support protocolaire équivalente.
- Nous avons évalué les performances à l'exécution de différentes couches de support protocolaire dans des conditions réelles. Nous avons constaté à robustesse équivalente aucune perte de performance significative.

L'approche fondée sur l'utilisation des langages dédiés que nous proposons dans cette thèse ouvre de nouvelles perspectives quant à l'utilisation de langages dédiés pour le développement d'applications réseaux robustes et performantes.

Mots clés

Langages dédiés, applications réseaux, protocoles textuels, génie logiciel

Abstract

A network application communicates with other applications according to a set of rules known as a protocol. This communication is managed by the part of the application known as the protocol-handling layer. This protocol-handling layer enables the manipulation of protocol messages. This layer is a critical component of a network application since it represents the interface between the application and the outside world. It must thus satisfy two constraints : it must be efficient to be able to treat a large number of messages and it must be robust to face various attacks targetting the application itself or the underlying platform. Despite these constraints, the development process of this protocol-handling layer still remains rudimentary and requires a high level of expertise. It consists of translating the protocol specification written in a high level formalism such as ABNF towards low level code such as C. The gap between these abstraction levels can entail many errors.

This thesis proposes a new language-based approach to the development of protocol-handling layers, to improve their robustness without compromising their performance. Our approach is based on the use of a domain-specific language, Zebu, to specify the protocol-handling layer of network applications that use textual application protocols *à la* HTTP. The Zebu syntax is very close to that of ABNF, facilitating the adoption of Zebu by domain experts. By annotating the original ABNF specification of a protocol, the Zebu user can dedicate the protocol-handling layer to the needs of a given application. The Zebu compiler first checks the annotated specification for inconsistencies and then generates a protocol-handling layer according to the provided annotations. This protocol-handling layer is made of a set of data structures that represent a message, a parser that fills in these data structures and various stub functions to access these data structures or drive the parsing of a message. By default, the generated message parser only accepts messages that strictly conform to the protocol. This validation criteria can be relaxed for more flexibility and better performance.

The contributions of this thesis are as follows :

– We carry out a complete analysis of the network application domain. Guided by the results of this domain analysis, we identify the input parameters of the design of a domain-specific language.
– We present the design of a declarative domain-specific language, Zebu, for specifying a protocol-handling layer dedicated to a given application. We have developed various specifications to validate the expressiveness of the language.
– We describe a Zebu compiler that generates C code that can be used with the Linux operating system. We experimentally assess the robustness of protocol-handling layers generated using the Zebu approach, and find that four times more invalid messages are detected by a Zebu-based protocol-handling layer than by equivalent protocol-handling layer.
– We perform performance experiments on various protocol-handling layers in real conditions, and find that the use of Zebu incurs no significant performance penalty for an equivalent degree of robustness.

The approach based on the use of domain-specific languages that we propose in this thesis opens up new possibilities for the development of the robust and efficient network applications.

Key words

Domain-Specific Languages, Network applications, Text-based protocols, Software Engineering

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Les communications entre applications s'effectuent par le biais de protocoles, ensembles consensuels d'un format ou structure de messages et de règles régissant les échanges de ces messages. Dans un premier temps dédiés aux communications machine à machine, le niveau d'abstraction de ces protocoles s'est élevé pour se rapprocher des applications et ainsi devenir des protocoles applicatifs, proches de l'utilisateur. L'avènement du protocole HTTP pour le transfert de pages Internet illustre cette montée en abstraction. En effet, le protocole HTTP est extensible par construction pour pouvoir intégrer des spécificités propres à une application. Le succès de ce protocole a inspiré la conception de nouveaux protocoles applicatifs. Ces protocoles sont conçus autour des caractéristiques principales du protocole HTTP à savoir le mode d'interaction fondé sur l'échange de requêtes d'un client vers un serveur et de réponses dans le sens opposé, l'encodage textuel par opposition à un encodage binaire, la structure des messages échangés et l'extensibilité du protocole par construction.

L'ouverture des applications au monde extérieur a eu pour conséquences de les exposer à des exigences de performance et à des menaces d'attaques qui leur étaient étrangères. Une application communicant grâce à un réseau, ou application réseau, se divise en deux couches, une couche de support protocolaire et une couche de logique applicative. La couche de support protocolaire gère la manipulation des messages : l'analyse syntaxique des messages reçus et la modification ou la construction des messages à transmettre. Cette couche représente l'interface entre l'application à proprement parler, c'est-à-dire la logique applicative, et le monde extérieur. Par conséquent, cette couche est l'ultime rempart de l'application contre les attaques extérieures. Faire céder ce rempart par une attaque permet un accès non contraint à la logique de l'application et potentiellement à la plate-forme matérielle sous-jacente[QPP02]. Ce rempart doit donc être le plus robuste possible. Parallèlement à cette robustesse, certaines applications doivent se montrer très réactives selon, par exemple, la tâche qu'elles ont à accomplir ou leur positionnement dans le réseau. Par réactivité, nous entendons la capacité de l'application à assurer un débit de messages important tant en entrée qu'en sortie. Par voie de conséquence, la couche de support protocolaire doit être efficace pour atteindre ce débit important.

L'omniprésence d'infrastructures de communication et la démocratisation du réseau Internet ont conduit à l'émergence de nouveaux domaines applicatifs tels que la téléphonie sur IP ou la vidéo à la demande. Avec l'émergence de ces nouveaux domaines applicatifs, de nouvelles applications sont apparues reposant sur de nouveaux protocoles applicatifs. Bien que de nouveaux protocoles apparaissent et soient standardisés, et que chaque protocole serve de support à une multitude d'applications aux besoins divers et variés, peu ou pas de progrès ont été faits pour améliorer le processus de développement du support protocolaire d'applications réseaux dans le cadre de protocoles applicatifs.

La difficulté principale du développement du support protocolaire d'applications réseaux dans ce

cadre est l'effort de traduction nécessaire entre la spécification formelle du protocole et le programme réel à implanter sur une plate-forme matérielle. Ainsi, le développement d'une application réseau demande une triple expertise : expertise de la logique applicative, expertise du protocole utilisé, expertise de programmation. Cependant, il est rare de disposer de ces trois expertises. De plus, du point de vue de la logique applicative, la manière dont sont structurées les données n'a pas d'importance. En effet, un protocole représente seulement la manière dont sont *sérialisées* les données qui vont transiter sur le réseau. Ainsi, reposer sur un protocole applicatif ne doit pas être un frein au critère de qualité logicielle qu'est l'*accessibilité au développement*. Dans notre cas, nous souhaitons que le concepteur de la logique applicative puisse conduire de manière autonome le processus de développement d'une application réseau dans son ensemble.

1.1 Thèse

La thèse présentée dans ce document propose une approche d'aide au développement du support protocolaire d'applications réseaux. Cette approche repose sur l'introduction d'un langage dédié à ce domaine permettant la génération d'une couche de support protocolaire robuste et efficace.

Un langage dédié à un domaine est un langage de programmation restreint à un domaine ou un problème particulier. Il est souvent déclaratif (par opposition à un langage procédural) pour masquer les détails d'implémentation à l'utilisateur. La spécificité d'un langage à un domaine se matérialise par l'introduction d'abstractions et de notations spécifiques à ce domaine. La restriction à ce même domaine résulte en une restriction du pouvoir d'expression du langage. Le contrôle de l'expressivité d'un langage facilite l'analysabilité du langage et permet ainsi la détection d'erreurs tôt dans le processus de développement.

Notre approche se fonde sur l'introduction d'un langage dédié à la spécification du support protocolaire d'applications réseaux, nommé Zebu. Ce langage, fondé sur un des formalismes utilisés pour la spécification de protocoles, permet le développement rapide d'une couche de support protocolaire, robuste et efficace, adaptée aux besoins d'une logique applicative particulière. Le traitement d'une spécification Zebu s'effectue en deux étapes. Premièrement, la spécification est analysée afin de déceler d'éventuelles incohérences. Ensuite, la couche de support protocolaire pour la manipulation de messages est générée automatiquement. Des outils attenants au compilateur permettent une intégration complète du code généré dans une infrastructure existante.

1.2 Contributions

Les contributions de cette thèse peuvent être classifiées en deux volets. Dans un premier temps, nous montrons la faisabilité et l'intérêt d'une approche langage dans des contextes fortement contraints tant en termes de robustesse que de performance. Nous présentons le processus complet de conception d'un langage dédié à la spécification du support protocolaire d'applications réseaux : depuis les analyses préalables jusqu'à l'implémentation du langage. Dans un second temps, nous validons notre approche par l'évaluation du langage Zebu.

Analyses de domaine et de famille de programmes. Nous présentons une analyse complète fondée sur l'étude des applications réseaux reposant sur des protocoles à la HTTP et plus particulièrement du support protocolaire associé. Nous mettons en avant les contraintes fortes et les défis à relever pour ce support. Nous mettons en évidence les points communs et les variations observées sur ces différentes applications formant les membres d'une même famille de programmes. Nous définissons les objets de

base du domaine. Enfin, nous présentons un cahier des charges définissant les opérations, relations et contraintes des objets du domaine.

Zebu. Nous introduisons un langage dédié, nommé Zebu, pour spécifier le support protocolaire associé à une application réseau. Dans la lignée de langages dédiés au traitement de données structurées, Zebu repose sur le formalisme utilisé pour la spécification de protocoles, permettant de réduire l'effort de programmation tout en introduisant des abstractions offrant des opportunités de vérifications de cohérence. Nous présentons le compilateur ZebuYacc et la chaîne de compilation optimisée μ-Zebu. Nous décrivons également le processus de développement d'une application réseau reposant sur notre approche.

Robustesse et efficacité. Nous présentons une évaluation de l'impact de l'utilisation de Zebu sur la robustesse et l'efficacité du support protocolaire d'une application réseau. Les résultats des expériences que nous avons menées sur des applications représentatives montrent que le support protocolaire fondé sur Zebu détecte 100% des messages erronés reçus contre environ 25% pour les couches de support protocolaire développées manuellement. De plus, le support protocolaire généré s'avère, à robustesse équivalente, aussi efficace que du support protocolaire développé manuellement.

1.3 Présentation 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 l'évaluation de notre solution.

Contexte. La première partie porte sur l'étude du contexte dans lequel nous nous plaçons. Les chapitres 2 et 3 présentent les deux composantes fondamentales de notre travail : les communications réseaux et les langages dédiés. Le chapitre 4 présente certains travaux significatifs dans le domaine considéré. Dans le chapitre 5, nous présentons la démarche que nous avons adoptée pour répondre aux problèmes posés par le développement du support protocolaire d'applications réseaux. Cette démarche décrit notre approche partant de la problématique originelle jusqu'à l'évaluation de l'approche proposée.

Approche proposée. Dans la deuxième partie, nous décrivons les différentes phases du développement de notre approche. Le chapitre 6 présente les analyses de domaine et de famille de programmes menées pour définir les différentes entrées de la conception de notre langage. Le chapitre suivant présente le langage Zebu à proprement parler pour la génération de la couche de support protocolaire adapté à une logique applicative particulière. Enfin, dans le chapitre 8, nous présentons le développement d'applications réseaux fondé sur le langage Zebu.

Évaluation. Dans la dernière partie de ce document, nous évaluons notre implémentation du langage Zebu. Le chapitre 10 présente une évaluation de l'impact de Zebu sur la robustesse du support protocolaire d'applications réseaux. Nous fondons notre évaluation sur un dérivé d'une technique appelée *analyse de mutations* qui nous permet, en envoyant des messages particuliers, d'évaluer la couverture du support protocolaire par rapport à la spécification de ce protocole. Enfin, nous présentons, dans le chapitre 10, une évaluation des performances à l'exécution du support protocolaire généré dans le cas de plusieurs applications significatives et ce, dans des conditions réelles.

Enfin, le chapitre 11 conclut en récapitulant nos différentes contributions et présente diverses perspectives de recherche.

Première partie

Contexte

Chapitre 2

Communications réseaux

Dans son acceptation courante, l'informatique désigne l'ensemble des techniques de traitement et d'échange de l'information. Cet échange se fait grâce aux réseaux qui relient les équipements informatiques afin qu'ils puissent communiquer entre eux. Or, pour communiquer, deux entités distinctes ont besoin d'un ensemble consensuel de conventions. Cet ensemble est composé de la définition d'un format des messages échangés et de règles régissant les échanges de ces messages. Ce format et ces règles constituent un protocole.

Néanmoins, un seul protocole ne peut englober les nombreux aspects d'une communication. Ainsi, par exemple, il faut gérer la retransmission des paquets perdus dans le contexte d'une transmission sans-fil. Un autre aspect est l'acheminement correct de l'information entre les entités concernées, seul le destinataire originel devant recevoir l'information transmise. À cause de la multiplicité des aspects d'une communication à gérer, les modèles de communication informatique se présentent en une succession de couches logiques. Chaque couche représente un aspect de la communication et à chaque couche correspond un protocole.

Dans ce chapitre, nous présentons les communications réseaux. Suite à une introduction sur les réseaux informatiques, nous nous focalisons sur les protocoles. Nous séparons les protocoles des couches inférieures de ceux des couches supérieures, pour finalement présenter les applications réseaux.

2.1 Réseaux informatiques

Les réseaux informatiques sont nés du besoin de relier des équipements informatiques pour qu'ils puissent communiquer entre eux. Le premier réseau informatique commercial, SABRE [Eva67], reliait ainsi mille deux cent téléscripteurs pour la gestion des réservations des vols de la compagnie aérienne American Airlines.

2.1.1 Introduction

Un réseau informatique est un réseau numérique. Un réseau numérique transfère des paquets de données d'une de ses extrémités à une autre par le biais d'entités appelées nœuds. Sur le *medium* physique de communication (*e.g.* un câble), les données correspondent à une série de `0` et de `1` (des *bits*) non interprétés, on parle aussi de flots de bits. Ces paquets de données correspondent à un découpage d'un message en plus petits éléments pour faciliter leur acheminement. Une fois tous les paquets de

données reçus par le destinataire final, le message est ré-assemblé pour délivrer l'information associée, c'est-à-dire l'interprétation sémantique de ces données.

Les réseaux informatiques sont classifiés en quatre catégories selon la distance maximale entre les points les plus éloignés du réseau. Un réseau personnel interconnecte sur quelques mètres les équipements d'un même utilisateur. Un réseau local permet d'échanger de l'information à l'échelle d'un bâtiment. Un réseau métropolitain est un réseau spécialisé, géré à l'échelle d'une métropole. Enfin, un réseau étendu est destiné à transporter des données sur de très grandes distances. Une problématique induite par la taille croissante de ces réseaux est l'hétérogénéité des réseaux. En effet, il existe différentes technologies qui permettent de véhiculer des données, ne serait-ce qu'au niveau matériel. Par exemple, des réseaux filaire, sans-fil et satellitaire n'ont pas les mêmes contraintes physiques ni les mêmes objectifs. Pourtant, dans le contexte d'un réseau étendu, il faut pouvoir les faire interagir.

Un réseau informatique a pour but de véhiculer de l'information entre des applications. Outre le volet d'échanges de messages et des règles inhérentes à cet échange, il est nécessaire de structurer la communication au niveau architectural. Les modes de communication remplissent cette tâche en s'appuyant sur les notions de client et de serveur.

2.1.2 Mode de communication

Les réseaux informatiques sont fondées sur les notions de client et de serveur. Le client demande à un serveur d'accomplir un service. Ensuite, l'échange de données entre ce client et ce serveur suit un certain mode de communication.

Le mode client/serveur consiste à distinguer un ou plusieurs clients et un serveur. Chaque client envoie des requêtes à ce serveur. À chaque requête reçue, le serveur effectue un traitement et peut renvoyer une réponse, on dit qu'il rend un service. Ce mode est aussi appelé mode maître/esclave dans le sens où le serveur est l'esclave du client puisqu'il obéit à ses requêtes.

Ce mode est historiquement le premier à être apparu. Ainsi, les technologies qui le soutiennent sont plus matures que les technologies plus récentes, telles que les technologie pair à pair par exemple. Un autre avantage du mode client/serveur est que toutes les données sont centralisées. Cette centralisation au niveau d'un serveur facilite l'administration des données et leurs mises à jour. Néanmoins, cette centralisation peut représenter un handicap majeur car un serveur devient la cible privilégiée d'attaques.

2.1.3 Applications

Dans le mode client/serveur, les applications qui communiquent peuvent être classées en quatre catégories : client, serveur mandataire ou *proxy*, passerelle et serveur. Ces applications se distinguent par la fonction qu'elles vont remplir. Le client, ou l'application cliente, va initier un dialogue avec un serveur. Le serveur mandataire agit comme un relais entre un client et un serveur. Sa position intermédiaire lui permet de remplir certaines fonctions comme l'anonymisation ou l'équilibrage de charge entre différents serveurs. La passerelle est une application un peu particulière dans le sens où elle ne fait qu'adapter les messages qui transitent entre deux réseaux hétérogènes. Le serveur est un programme, et par abus de langage un équipement, qui rend un service. Le plus connu est le serveur Web, notamment httpd de la fondation Apache [Apa], qui est contacté par un client pour fournir des pages Internet. Néanmoins, quel que soit le mode de communication utilisé, l'échange de l'information se fait toujours grâce à des messages. Une communication est régie par un ensemble de règles et un format de messages à manipuler.

2.1.4 Messages

Les réseaux informatiques avaient, dans un premier temps, vocation à relier des terminaux distants à un site central, puis des ordinateurs entre eux et enfin des machines terminales, telles que des stations de travail. Nous sommes passés d'une communication exclusivement *machine à machine* vers une communication *humain à machine* avec l'introduction d'un nouvel élément, l'utilisateur humain. Un utilisateur n'a pas les mêmes capacités qu'une machine. Un humain aura par exemple beaucoup plus de difficultés à extraire de l'information d'un flot d'octets que d'un texte structuré.

Simultanément, les équipements informatiques ont disposé d'une puissance de calcul accrue. Historiquement, les réseaux informatiques se distinguaient des autres réseaux numériques (entre autres certains réseaux téléphoniques) par le fait que l'intelligence se situait aux extrémités du réseau, au niveau des équipements terminaux. Les terminaux gèraient et contrôlaient le réseau. Par exemple, pour qu'il n'y eût pas d'embouteillage de paquets dans le réseau, un terminal s'auto-régulait. Avec l'amélioration des performances des équipements réseaux, de l'intelligence peut être injectée à l'intérieur même du réseau, notamment au niveau de serveurs mandataires. Ces équipements réseaux deviennent donc programmables.

La nature des messages échangés a été fortement influencée par ces évolutions. Dans un premier temps, les communications exclusivement machine à machine avec des ressources limitées ont contraint l'utilisation de messages fondés sur un encodage binaire. Un encodage binaire consiste à décrire l'information dans un format proche de la machine, en codant l'information sous forme de bits. Cet encodage présente l'avantage principal d'être concis, ce qui permet de limiter la taille des échanges nécessaires sur le réseau et également de minimiser les ressources nécessaires à l'analyse propre du message (en dehors de toute phase de traitement inhérente à l'information véhiculée).

Avec l'essor des communications humain à machine et l'augmentation de puissance des équipements et des réseaux, l'encodage binaire est devenu un choix plus qu'une nécessité. Ainsi, certains échanges se fondent sur un encodage textuel. Avec cet encodage, les messages circulent *en clair* sur le réseau. Outre un développement facilité pour le programmeur car la manipulation de texte est plus naturelle que la manipulation de *bits*, l'encodage textuel permet une mise au point plus aisée.

Si le format des messages binaire peut être défini à l'aide de ce que l'on appelle un masque de *bits*, il en va différemment pour les protocoles textuels. Le format des messages doit être spécifié par un formalisme adapté à la définition de grammaire. Ainsi, la grammaire qui décrit le format des messages, est formalisée à l'aide de la *metasyntaxe* Augmented Backus-Naur Form [CO97] (ABNF) . L'ABNF est dédiée à la spécification de syntaxe des messages. Elle dérive de la *metasyntaxe* Backus-Naur Form [Knu64] (BNF) qui est utilisée pour la définition de grammaires de langages de programmation. Il en résulte quelques différences syntaxiques mineures.

Les messages textuels peuvent également être structurés à l'aide du langage XML [BPSM+06] pour capitaliser sur les outils développés dans le cadre de l'utilisation de XML pour la persistance de données. L'intérêt de l'utilisation de XML est que la grammaire des messages peut être spécifiée formellement sous la forme d'une Document Type Definition [dtda](DTD) ou d'un schéma XML [dtdb]. Des outils idoines vont pouvoir s'appuyer sur ces grammaires pour garantir que les messages sont bien formés et valides.

Dans cette section, nous avons introduit la notion de réseau informatique en définissant quatre catégories selon la taille du-dit réseau. L'adoption d'un mode de communication particulier garantit le bon fonctionnement du réseau en définissant une certaine structuration des communications. Dans le contexte du mode client/serveur, nous avons défini les quatre classes d'application. Ces applications, pour communiquer, échangent des messages binaires ou textuels dont certains fondés sur le langage XML. Néanmoins, s'accorder sur un format de message est une condition nécessaire mais

pas suffisante à la communication entre entités. Un protocole réseau organise et structure l'intégralité de la communication entre deux entités distinctes en définissant un format de messages et les règles d'échange de ces messages.

2.2 Protocoles réseaux

Le but de la communication entre entités, nous l'avons vu, est l'échange de l'information. Dépourvues de structuration et de règles régissant cet échange, les données véhiculées seraient inexploitables. Elles demeureraient une succession de bits impossible à interpréter.

Un protocole de communication est un consensus de communication entre deux entités. Il regroupe une structuration des messages et des règles régissant l'échange de ces messages. Ces règles définissent une machine à états qui selon la nature du message reçu décrit les messages à transmettre.

La notion de protocole provient du monde des télécommunications, son illustration la plus simple étant celle de la conversation téléphonique. Un émetteur qui souhaite transmettre une information compose le numéro de téléphone du récepteur souhaité. Ce récepteur entendant la sonnerie, décroche et prononce le mot "Allô". Ce mot est un signal à destination de l'émetteur indiquant que son interlocuteur est prêt à recevoir de l'information. À la fin de leur conversation, les deux interlocuteurs se mettent d'accord sur le fait qu'ils n'ont plus rien à transmettre, puis ils raccrochent.

Nous voyons ici qu'il y a deux niveaux de communication ; le niveau inférieur avec la communication entre les combinés téléphoniques (décrocher/raccrocher) et le niveau supérieur avec la communication entre les deux interlocuteurs. Le niveau inférieur supporte et englobe le niveau supérieur de communication. L'établissement d'une communication au niveau inférieur est un pré-requis obligatoire pour l'établissement de la communication au niveau supérieur.

2.2.1 Modèles

La structuration en couches des protocoles de communication a été standardisée par la norme ISO 7498 de l'Organisation de Standardisation Internationale (International Standard Organisation). Cette norme intitulée *Modèle basique d'interconnexion de systèmes ouverts* est le plus souvent référencée sous le nom de modèle OSI, d'après son titre en langue anglaise (*Open System Interconnection Model*). Chaque couche est articulée autour de trois notions qui sont la notion de service, celle de protocole et celle de point d'accès au service : 1) le service est la description abstraite de fonctionnalités telles que la demande de connexion ou la réception de données ; 2) Le protocole est un ensemble de messages et de règles d'échanges réalisant un service ; 3) Le point d'accès au service est le moyen concret d'utiliser le service. Chaque protocole encapsule dans son propre message le message du protocole de la couche immédiatement supérieure.

Le modèle OSI définit sept couches comme le montre la partie gauche de la figure 2.1. Les quatre couches *basses* sont orientées communication et les trois couches *hautes* sont orientées application.

Ce modèle n'est pas utilisé dans la pratique à cause de certaines limitations inhérentes à sa spécification. Ainsi, le modèle OSI prévoit l'utilisation d'un seul protocole par couche, ce qui est impossible dans le contexte d'interconnexion de réseaux hétérogènes. De plus, sa spécification est parue trop tardivement par rapport aux besoins de communication.

Un modèle *pratique*, dit modèle TCP/IP était déjà en place, basé sur les protocoles du même nom : *Transmission Control Protocol* [Pos81b] et *Internet Protocol* [Pos81a]. La partie droite de la figure 2.1 représente ce modèle et quelques protocoles pour chaque couche. Il existe une correspondance des couches basses entre les deux modèles, comme le montre la figure 2.1. Néanmoins, le

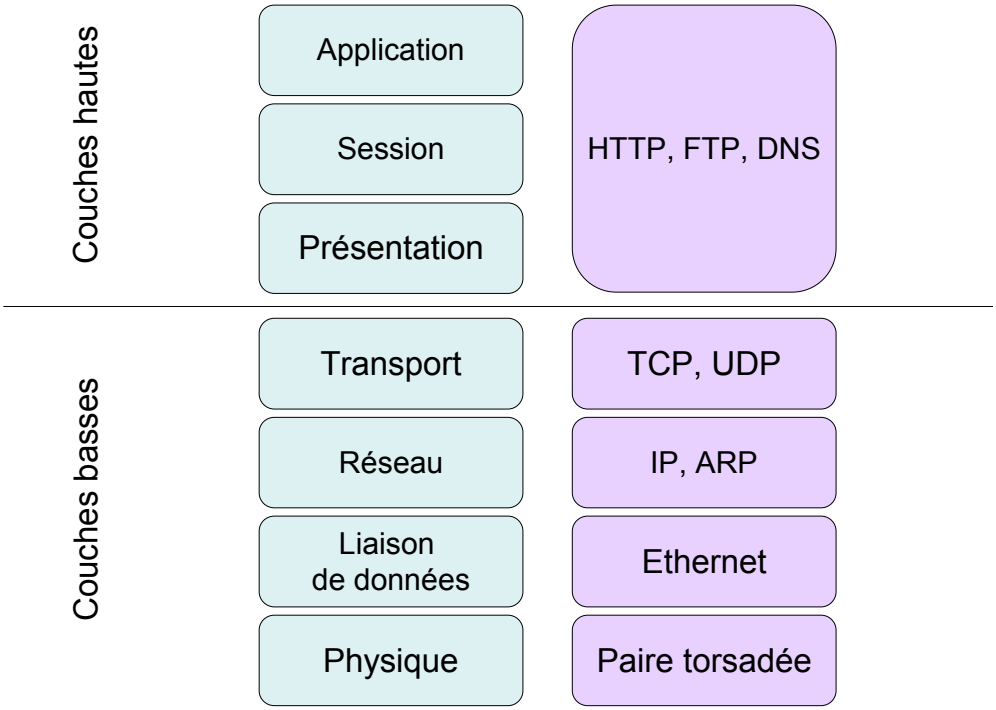

FIG. 2.1 – Couches du modèle OSI de l'ISO et protocoles associés dans le modèle TCP/IP

modèle IP prend quelques latitudes par rapport à la norme de l'ISO. Par exemple, le modèle OSI garantit l'indépendance entre deux couches successives. Cependant, certains protocoles du modèle IP utilisent de l'information de protocoles sous-jacents. Ainsi, le protocole UDP [Pos80] utilise de l'information de la couche IP pour calculer une somme de contrôle utilisée pour la détection d'erreurs. Les couches hautes des deux modèles diffèrent largement. Les couches session et présentation sont définies de manière approximative dans le modèle OSI. La couche session permet de définir des sessions d'échange avec des points de synchronisation. Si une session est interrompue, les données reçues sont conservées et l'échange peut reprendre à partir d'un point de synchronisation. La couche présentation permet d'effectuer les derniers réglages de mise en forme des données pour l'application. Ces deux couches, session et présentation, disparaissent dans la pratique au profit d'une couche unique, la couche application.

Nous détaillons maintenant les différentes couches du modèle pratique TCP/IP en distinguant les couches *basses* de la couche *haute*.

2.2.2 Couches basses du modèle TCP/IP

Les quatre couches basses ont pour but d'identifier et d'atteindre une entité distante. Nous sommes ici dans un contexte de communication machine à machine, les messages échangés utilisent généralement un encodage binaire (pour des raisons historiques également).

La première tâche à réaliser quand une entité souhaite communiquer sur un support réseau (*medium*) est l'adaptation des données à ce support. Il faut par exemple convertir les 0 et 1 en signaux lumineux pour l'utilisation d'une fibre optique et inversement. Cette adaptation est la responsabilité de

la couche physique. Ensuite, il faut pouvoir identifier et atteindre un équipement voisin à l'intérieur d'un même réseau local. Cet équipement peut servir de relais vers un autre réseau. Cette phase est prise en charge par la couche liaison de données. L'étape suivante dans la communication consiste à sortir de son réseau local pour établir une communication bout à bout, à travers différents réseaux. Un protocole de la couche réseau prend en charge le routage qui consiste à déterminer la route pour atteindre la destination finale. Dans ce contexte, il est nécessaire pour un équipement d'avoir une adresse logique qui l'identifie de manière unique. Cette adresse logique unique est fournie par la couche réseau par l'entremise du protocole IP par exemple. Nous parlons ici d'adresse logique par opposition à la méthode d'adressage dite physique de la couche liaison de données. L'adresse logique permet de savoir dans quel réseau se situe l'équipement contrairement à l'adresse physique qui n'est qu'un numéro de série. Cette adresse physique n'a de sens que dans le cadre de communications à l'intérieur d'un même réseau local. Enfin, comme nous l'avons dit précédemment, le but des communications réseaux est de faire communiquer des applications. La couche réseau permet seulement d'atteindre un équipement distant. Or plusieurs applications peuvent s'exécuter sur une même machine. La couche transport veille à ce que les messages soient transmis à l'application concernée.

Les quatre couches basses du modèle OSI garantissent que toutes les données nécessaires à la restitution de l'information sont présentes. Le support de la communication est dit fiable s'il ne peut être mis en défaut par rapport à la spécification. Le support de communication est dit sûr si quelque soit le message reçu, son comportement est garanti.

Dans le but d'avoir une certaine universalité de ce support basal, tous les protocoles utilisés dans ces couches sont standardisés notamment par l'Internet Engineering Task Force. Une caractéristique importante de ces protocoles est qu'ils sont très stables, pour ainsi dire figés, depuis près de quarante ans pour certains. Le développement du support logiciel de ces protocoles s'est fait manuellement, faute d'outils de génération automatique performant. Néanmoins, au cours de ces nombreuses années, ce support logiciel a été largement éprouvé. La stabilité des spécifications a entraîné l'intégration de ce support logiciel des couches basses dans tous les systèmes d'exploitation modernes. Cette intégration a contribué à le rendre très fiable, sûr et efficace.

2.2.3 Couche haute du modèle TCP/IP

Les couches hautes du modèle OSI sont fusionnées en une seule couche application dans le modèle TCP/IP. En effet, le modèle pratique est dérivé d'implémentations concrètes qui avaient fusionné les couches présentation, session et application en une seule couche application.

Cet ultime niveau de communication s'établit entre deux applications. Le protocole véhicule des paramètres pour ces programmes. Les applications qui communiquent sont apparentées dans le sens où elles se situent dans un même domaine d'application. Un domaine d'application ou domaine applicatif correspond à un ensemble d'applications reliés par une fonctionnalité commune. De fait, ces protocoles ne sont pas génériques mais spécifiques à des domaines d'application particuliers. Nous pouvons citer Single Mail Transfert Protocol [Pos82] (SMTP) pour l'acheminement d'e-mails, Session Initiation Protocol [Ros02] (SIP) pour la téléphonie sur IP, ou encore Simple Network Management Protocol [CFSD90] (SNMP) pour la supervision d'équipements sur un réseau. Cette spécificité à un domaine d'applications voire même à une seule application induit la non-systématicité de la standardisation de ces protocoles. Par exemple, il n'est pas forcément nécessaire de standardiser un protocole régissant les communications entre équipements ayant un besoin très marginal, par exemple dans les réseaux de terrain [Tho99].

Contrairement aux protocoles de couches basses, ces protocoles évoluent ou sont étendus. De nouveaux protocoles apparaissent également avec l'émergence de nouveaux domaines applicatifs.

Aujourd'hui, le développement du support logiciel pour ces protocoles applicatifs est exclusivement manuel. Or, les changements constants et la complexité des protocoles rend ce support peu fiable. La recherche d'efficacité, également, fragilise ce support logiciel en le rendant plus complexe et par conséquent plus propice aux erreurs et moins maintenable.

2.2.4 Illustration

Nous avons vu dans la section 2.1.4 les différents encodages possibles pour des messages protocolaires. Cette notion d'encodage est théoriquement orthogonale à la notion de couche préalablement définie. Néanmoins, nous revenons sur cette notion car elle détermine le format des messages en l'illustrant par des exemples concrets de protocoles à la lumière des sections précédentes.

Protocoles binaires L'encodage binaire consiste à coder l'information en bits. De nombreux *anciens* protocoles utilisent cet encodage, tel TCP [Pos81b], UDP [Pos80] et IP [Pos81a]. L'encodage binaire ne se cantonne pas aux couches basses, des protocoles haut niveau peuvent également adopter cet encodage à l'image de H.323[MMS00] pour la téléphonie.

Protocoles XML Le langage XML est un support émergent pour les formats des messages dans le contexte des communications réseaux. Dans le sillage de son utilisation massive dans la représentation de données structurées persistantes, il s'est imposé comme une alternative intéressante dans l'échange de l'information. Outre son usage le plus connu dans les services Web [BHM+04], certains protocoles récents reposent sur ce langage. C'est le cas, notamment, du protocole eXtensible Messaging and Presence Protocol [SA04] (XMPP), protocole de messagerie instantanée utilisée par GoogleTalk [Goo] dont un message est représenté à la figure 2.2.

```
<message
     to='romeo@example.net'
     from='juliet@example.com/balcony'
     type='chat'
     xml:lang='en'>
   <subject>I implore you!</subject>
   <subject xml:lang='cs'>
     &#x00DA;p&#x011B;nliv&#x011B; pros&#x00EDm!
   </subject>
   <body>Wherefore art thou, Romeo?</body>
   <body xml:lang='cs'>Pro&#x010D;e&#x017E; jsi ty, Romeo?</body>
 </message>
```

FIG. 2.2 – Exemple de message XMPP

L'avantage principal de l'utilisation de XML dans ce contexte est de pouvoir capitaliser sur tous les outils préalablement développés comme la librairie Castor [Gro] ou le langage Xduce [HP00] conçu pour manipuler et traiter des données XML (on parle de document XML ou de message XML).

L'inconvénient majeur de l'utilisation de XML dans les communications réseaux est qu'historiquement les outils de support à XML ciblaient le traitement de données hors-ligne, sans contrainte d'efficacité en temps et en mémoire. Au delà de la lenteur relative de ces outils, les systèmes contraints en ressources tels que les systèmes embarqués, sont ségrégués de l'utilisation de ces outils. Plus généralement, les systèmes embarqués n'utilisent pas de communication à base de XML même si certains travaux existent [Micb].

Protocoles textuels Les protocoles textuels comme HyperText Transfert Protocol (HTTP) ou SIP (dont un message est représenté en figure 2.3) restent populaires car ils allient un développement plus naturel, une mise au point aisée et une utilisation possible même dans des systèmes contraints. La figure 2.4 représente un extrait de l'ABNF de SIP définissant l'en-tête `Via` (deuxième ligne de la figure 2.3).

```
INVITE sip:bob@biloxi.example.com SIP/2.0
Via: SIP/2.0/TCP client.atlanta.example.com:5060;branch=z9hG4bK7
Max-Forwards: 70
Route: <sip:ss1.atlanta.example.com;lr>
From: Alice <sip:alice@atlanta.example.com>;tag=9fxced76sl
To: Bob <sip:bob@biloxi.example.com>
Call-ID: 3848276228511@atlanta.example.com
CSeq: 1 INVITE
Contact: <sip:alice@atlanta.example.com;transport=tcp>
Content-Type: application/sdp
Content-Length: 151

v=0
o=- 992124952284092 992124952284099 IN IP4 147.210.177.85
s=Media Presentation
e=NONE
c=IN IP4 0.0.0.0
[...]
```

FIG. 2.3 – Exemple de message SIP

```
Via                  = ( "Via" / "v" ) HCOLON via-parm *(COMMA via-parm)
via-param            = sent-protocol LWS sent-by *( SEMI via-params )
via-params           = via-ttl / via-maddr / via-received / via-branch / via-extension
via-ttl              = "ttl" EQUAL ttl
via-maddr            = "maddr" EQUAL host
via-received         =  "received" EQUAL (IPv4address / IPv6address)
via-branch           =  "branch" EQUAL token
via-extension        =  generic-param
sent-protocol        =  protocol-name SLASH protocol-version SLASH transport
protocol-name        =  "SIP" / token
protocol-version     =  token
transport            =  "UDP" / "TCP" / "TLS" / "SCTP" / other-transport
sent-by              =  host [ COLON port ]
ttl                  =  1*3DIGIT ; 0 to 255
```

FIG. 2.4 – Extrait de l'ABNF de SIP pour l'en-tête Via

Néanmoins, cet encodage textuel a pour réputation d'être gourmand en ressources. Il n'existe, de surcroît, pas d'aide au développement logiciel pour le support de protocoles applicatifs textuels. Le support logiciel des ces protocoles est écrit à la main avec pour but premier l'efficacité. Néanmoins, comme nous l'avons dit précédemment, cette recherche de l'efficacité se fait au détriment de la fiabilité et de la sûreté du développement logiciel.

2.3 Applications réseaux

Nous avons vu dans la section 2.2.2 que le support des couches basses est fourni par le système d'exploitation. Par ailleurs, la section 2.2.3 a précisé que le support des protocoles de la couche application est à la charge du programmeur de l'application. Par conséquent, une application réseau se décompose en deux couches, la couche de support protocolaire et la couche de logique applicative. La couche inférieure abstrayant le protocole pour la couche supérieure qui met en œuvre la logique spécifique à l'application proprement dite. Dans la suite de ce document, nous nous concentrerons sur les applications réseaux s'appuyant sur des protocoles textuels.

2.3.1 Support protocolaire des applications réseaux

La couche de support protocolaire fournit une interface de manipulation des messages à l'application. Elle analyse le message pour en extraire l'information nécessaire à la couche de logique applicative. Toutefois, plusieurs contraintes sous-tendent sa programmation.

2.3.1.1 Contraintes

Les deux premières contraintes à prendre en compte lorsqu'on développe une couche de support protocolaire sont des contraintes fonctionnelles. La couche logicielle de support protocolaire aux protocoles textuels occupe une place critique dans l'architecture logicielle des applications réseaux. Elle est la cible de nombre d'attaques car elle représente l'interface entre l'application et le monde extérieur. Par conséquent, la couche de support protocolaire doit être robuste et efficace. La contrainte de robustesse présente deux aspects. D'une part, le support doit être complet par rapport à la spécification. Tous les cas spécifiés doivent être supportés. D'autre part, le support doit être correct. Un message invalide doit être détecté et une erreur doit être remontée. Une information erronée ne doit pas parvenir à la logique applicative. Évidemment, le support protocolaire doit également être efficace, à la fois en temps et en mémoire. La logique applicative ne doit pas être limitée en ressources à cause d'un support protocolaire déficient dans sa construction. La dernière contrainte est une contrainte *de facto*. Même si les protocoles sont formellement spécifiés dans des langages ou notations *ad hoc*, il n'existe pas d'outil consensuel permettant de générer automatiquement le support protocolaire d'applications réseaux. Le programmeur doit manuellement développer un support protocolaire qui soit robuste et efficace. Sa tâche n'est pas aisée, les spécifications de protocole étant généralement conséquentes et complexes.

2.3.1.2 Principes de programmation

Les contraintes mentionnées précédemment sous-tendent tout le développement du support protocolaire. En plus d'une charge de programmation importante due à la lourdeur des spécifications, le programmeur doit garantir la robustesse et l'efficacité du code développé.

La contrainte d'efficacité ainsi que certaines raisons historiques ont conduit à la généralisation de la programmation à l'aide de langages de bas niveau tel que le langage C [KD88]. L'écart des niveaux d'abstraction entre la spécification de la syntaxe du protocole en ABNF et le code à développer est important.

La mise en parallèle des figures 2.4 qui représente la grammaire de l'en-tête `Via` du protocole SIP et de la figure 2.5 qui représente le code de support protocolaire correspondant montre le fossé que doit combler le programmeur. La figure 2.5 représente un extrait du code du SIP Express Router [POJK03] (SER), un serveur mandataire SIP considéré aujourd'hui comme la référence en termes de support

protocolaire, à la fois concernant sa robustesse et son efficacité. L'analyse syntaxique de l'en-tête `Via` nécessite la mise en œuvre d'une machine à états à l'aide d'instructions d'aiguillage (*switch*) et de sauts inconditionnels (*goto*). Le traitement de l'en-tête `Via` compte pas moins de 2035 lignes comprenant 64 `switch`, 564 `case`, 232 `break` et 143 `goto`. L'oubli d'une seule de ces instructions rend le support protocolaire incomplet voire dangereux.

Des outils permettent toutefois de vérifier la conformité d'une implémentation par rapport à un modèle, on parle de *model-checking* [CGP99, Hol97]. Ces outils ont été utilisés dans la recherche pour vérifier l'implémentation d'application réseau et de pile de protocoles [ME04]. Néanmoins, nous ne connaissons pas d'application réseau majeure ayant subie une vérification complète de leur support protocolaire à l'aide de tels outils.

```
for (tmp=p;tmp<end;tmp++){
  switch(*tmp){
    case ' ':
      switch(state){
        case FIN_HIDDEN:
        case FIN_ALIAS:
          param->type=state;
          param->name.len=tmp-param->name.s;
          state=L_PARAM;
          goto endofparam;
        case FIN_BRAN
    CH:
        case FIN_TTL:
        case FIN_MADDR:
        case FIN_RECEIVED:
        case FIN_RPORT:
        case FIN_I:
          param->type=state;
          param->name.len=tmp-param->name.s;
          state=L_VALUE;
          goto find_value;
[...]
```

FIG. 2.5 – Partie du code permettant l'analyse de l'en tête *Via* dans SER

Les difficultés inhérentes à l'utilisation d'un langage de bas niveau ajoutées à la taille et à la complexité des spécifications de protocole rendent la mise au point du code du support protocolaire difficile et fastidieuse et ce code peu maintenable et évolutif.

2.3.1.3 Évaluation

Le protocole SIP est en passe de devenir le protocole incontournable de la téléphonie sur IP. Le logiciel iChat de Apple [ich] l'a déjà adopté et les prochaines moutures du client de messagerie de Microsoft s'appuieront sur ce protocole. De prime abord assez simple, ce protocole présente quelques particularités, spécifiques aux protocoles dérivés de HTTP, qui accroissent la complexité du code à écrire pour son support. Ces particularités conduisent à avoir un support protocolaire qui représente pas moins de 18% du code total d'un serveur mandataire SIP tel que SER. Certaines études récentes portent sur l'évaluation des performances du support protocolaire dans divers produits.

Une étude [CEE04] compare les performances de l'analyse syntaxique de quatre serveurs mandataires (S_A, S_B, S_C, S_D) aux caractéristiques générales différentes. Les serveurs S_A et S_B sont des implémentations simples utilisant les fonctions et types standards du langage C. L'analyse syntaxique qu'ils effectuent est complète, puisque tout le message est analysé avant de pouvoir être traité.

Les serveurs S_C et S_D sont optimisés, utilisant des fonctions et types dédiés. Ainsi, le serveur S_C s'appuie sur une infrastructure appelée Narnia [CE02] qui permet de développer des services de communication efficaces. Le serveur S_D fournit son propre module de gestion mémoire et des routines propriétaires optimisés. De plus, l'analyse syntaxique des ces deux serveurs est incrémentale ou étagée. Une première étape de l'analyse collecte les informations de position des éléments du message. La deuxième étape extrait et analyse les éléments nécessaires à la phase de traitement qui lui est concomitante. Les différentes expérimentations ont montré que près de 25% du temps de traitement complet du message est consacré à l'analyse syntaxique. Si les serveurs S_A et S_B ont des performances équivalentes, les différentes optimisations opérées sur les serveurs S_C et S_D permettent de diviser le temps de traitement complet du message par un facteur compris entre 4 et 10. Les résultats montrent également que l'augmentation des performances de l'analyse syntaxique est principalement obtenue grâce à son étagement. Une étude postérieure [WSKW07] a confirmé ces résultats pour SER dont l'analyse syntaxique est également étagée.

Le support protocolaire doit prendre en charge la détection de messages qui ne correspondent pas à la spécification et fonctionner correctement si un message est correct. Dans la pratique, les supports protocolaires sont beaucoup plus permissifs comme le préconise une recommandation de l'IETF, présente dans chaque spécification de protocoles qui conseille d'accepter les messages qui sont à la limite de la validité. Cependant, cette permissivité est souvent plus subie que réellement souhaitée, laissant la porte ouverte à de nombreuses attaques. La plate-forme multi-protocolaire de téléphonie Asterisk [Spe], largement déployée dans le monde de l'entreprise, est la cible de nombreuses attaques et notamment son support du protocole SIP. La réception par cette plate-forme d'un message SIP erroné permettait encore récemment de prendre le contrôle à distance de la machine avec les droits d'exécution de la plate-forme[1].

2.3.2 Logique applicative

La couche logicielle reposant sur la couche de support protocolaire implémente la logique de l'application ou logique applicative. Nous allons détailler dans la suite de cette section quatre catégories de logique applicative mais il est important de noter que ces logiques sont très fortement couplées, d'un point de vue fonctionnel, aux protocoles applicatifs sous-jacents. Les applications que nous considérons sont fondées sur le modèle client/serveur. Un client initie un dialogue en envoyant des requêtes à un serveur qui exécute un service. Durant l'exécution de ce service, un ou plusieurs échanges de requêtes et réponses peuvent avoir lieu. Ces échanges peuvent transiter via différents équipements qui peuvent agir au niveau applicatif pour réaliser, par exemple, du routage contextuel ou une adaptation de protocoles.

2.3.2.1 Client

Les butineurs, pour *browser*, Internet tels que Mozilla Firefox [Fou] ou Internet Explorer [Mica] sont des clients HTTP. Hormis l'adresse de la ressource recherchée, l'Uniform Resource Locator, l'utilisateur doit fournir des informations de connexion et de sécurité qui vont modifier la structure des messages envoyés. De la même manière, les clients SIP [Cou, Big] requièrent l'adresse de contact (l'équivalent du numéro de téléphone) d'un interlocuteur pour pouvoir l'appeler. Certains paramètres peuvent être nécessaires également pour l'authentification de l'appelant. Néanmoins, ces outils très proches d'un utilisateur n'ont pas vocation à être hautement configurables.

[1] source : http ://nvd.nist.gov/nvd.cfm ?cvename=CVE-2007-1306

D'autres clients permettent un contrôle plus fin sur la structure du message. Ils ne sont utilisés que dans des contextes particuliers. Par exemple, Seagull [sea] est un générateur de trafic multi-protocolaire pour valider les serveurs par rapport aux spécifications. Les messages envoyés sont déduits d'un fichier de configuration XML. Ainsi, il devient possible de soumettre les serveurs à des structures de message peu communes mais qui peuvent s'avérer problématiques.

2.3.2.2 Serveur mandataire ou *proxy*

Un usage récent, mais qui tend à se généraliser, est d'utiliser les serveurs mandataires comme des routeurs de niveau applicatif. Ils remplissent la même fonction que les routeurs du niveau réseau mais peuvent raisonner sur les messages du protocole applicatif. Dans le contexte du protocole HTTP, un *web switch*, d'après la définition donnée par Valeria Cardellini [CCCY02] examine les requêtes HTTP qui transitent par lui pour identifier le type de la ressource demandée (vidéo, audio, *etc.*). Généralement, pour identifier le type de la ressource, seul l'examen de l'URL est nécessaire. Suivant le type identifié, la requête sera dirigée vers un équipements dédié.

Concernant le protocole SIP, le serveur mandataire est chargé de la mise en relation de deux interlocuteurs. Il est généralement possible de lui adjoindre une politique de routage des appels comme dans SER avec des moyens programmatiques divers [BCL+06a, BCL+06b]. Cette politique définit principalement des critères de cheminement d'appels, par exemple en fonction de l'appelant.

2.3.2.3 Passerelle

La passerelle traduit une suite de messages d'un protocole dans la suite de messages correspondante dans un autre protocole, à l'instar des passerelles de niveau transport qui transfèrent les messages entre deux réseaux physiques. Certains protocoles reposent, en effet, sur le même domaine applicatif. Ce constat de la multiplicité de protocoles apparentés est particulièrement vrai dans des domaines émergents tels que la téléphonie sur IP [voi]. Néanmoins, au delà de différences syntaxiques simples, les protocoles ne sont pas strictement équivalents mais plutôt présentent une intersection suffisamment importante pour permettre une traduction.

La passerelle, telle que celle présentée par Hu entre SIP et H.323 [HHS01], extrait du message originel les éléments du message du protocole source qui peuvent être traduits. Suite à une phrase de traduction plus ou moins complexe, les éléments traduits dans le protocole cible sont insérés dans des patrons de message qui ont été statiquement définis.

2.3.2.4 Serveur

Les requêtes que le serveur reçoient sont finement analysées pour deux raisons. D'une part, contrairement au serveur mandataire et à la passerelle, le serveur doit analyser tout le message pour rendre son service. D'autre part, le serveur est un équipement fortement sollicité et donc cible de nombreuses attaques. Une analyse minutieuse des messages reçus est obligatoire pour détecter les messages malsains et les traiter en conséquence.

2.4 Bilan

Dans la première partie de chapitre, nous avons défini la notion de protocole. Nous avons introduit les modèles de catégorisation des protocoles fondés sur une séparation en couches fonctionnelles. Nous avons ensuite mis en opposition les couches basses et la couche haute du modèle pratique et leur

support logiciel. Si le support logiciel pour les couches basses est fourni par le système d'exploitation, le support de la couche haute est à la charge du programmeur de l'application.

Dans la seconde partie du chapitre, nous avons mis l'accent sur la programmation de cette couche haute. Nous avons divisé cette couche en deux sous-couches, le support protocolaire et la logique applicative. Le support protocolaire, l'interface entre l'application et le monde extérieur, doit faire face à des contraintes importantes de robustesse et d'efficacité. Ces contraintes sous-tendent tout le développement de ce support protocolaire. La logique applicative dont nous avons identifié quatre instances différentes repose sur ce support protocolaire pour s'exécuter. Il apparaît clairement que suivant la logique de l'application le support protocolaire nécessaire à l'exécution de cette logique n'est pas le même.

Chapitre 3

Langages dédiés

La programmation paraît souvent abstraite et comme insurmontable à des personnes qui ne sont pas familiarisées avec l'outil informatique en général et les langages de programmation en particulier. La raison principale est que les langages de programmation généralistes sont trop éloignés des considérations particulières d'utilisateurs non-programmeurs. Ces langages permettent de résoudre un grand nombre de problèmes mais ne prennent pas le débutant par la main pour le guider vers une solution particulière au problème donné. Par opposition, les utilisateurs sont au centre des préoccupations du concepteur de langage dédié. Le langage dédié permet de résoudre de manière simple et sûre certains problèmes particuliers de l'utilisateur.

Dans ce chapitre, après une définition succincte de ce qu'est un langage dédié, nous répondons à trois questions concernant les langages dédiés : *quand et pourquoi*, *quoi* et enfin *comment*. La première question est *quand et pourquoi*, à quel moment et pourquoi s'engager dans un processus de développement long et complexe et surtout pour quel gain. Une fois ce pas franchi se pose la question du *quoi* ou la conception du langage, que faut-il mettre dans un langage pour qu'il soit adopté par les utilisateurs du domaine et ne rejoigne pas de nombreux langages aux oubliettes. Enfin quand la conception est terminée, il reste la question du *comment*, c'est-à-dire comment implémenter un langage dédié et tous les outils qui peuvent s'avérer nécessaires à son utilisation. De bons outils sont nécessaires pour l'adoption du langage par les utilisateurs du domaine cible notamment en termes de détection et correction d'erreurs.

3.1 Introduction

Les termes *Domain-Specific Language* (DSL), langage métier ou langage dédié représentent un langage qui est spécifique à un domaine d'application particulier. Il est possible aussi de le définir par opposition aux langages généralistes (GPL pour *General Purpose Language*). En quelques mots, le langage dédié va permettre dans un contexte particulier de résoudre un problème par l'écriture d'un programme à la fois de manière beaucoup plus simple et beaucoup plus sûre qu'un langage généraliste.

Notamment, un langage dédié offre des abstractions et des notations utiles dans le domaine considéré. La cible d'un langage dédié est l'utilisateur du domaine considéré, rarement expert en programmation, à qui ces notations et abstractions doivent être familières. Ces éléments du langage vont accroître l'expressivité du langage au sens de Felleisen [Fel90] et la focaliser sur des concepts du domaine. Plus expressif signifie ici que traduire un programme écrit dans ce langage dédié avec ces abstractions et notations particulières vers un langage qui en est dépourvu nécessite une complète réorganisation du programme.

Un langage dédié peut être également restreint dans son pouvoir de calcul par un contrôle très fin des constructions mises à la disposition de l'utilisateur, pour des raisons de sûreté principalement. La restriction dans les constructions fait qu'il est souvent fait référence aux langages dédiés comme étant des *petits* langages. L'exemple typique reste la suppression de toute instruction de boucle non-contrôlée (telle l'instruction `while` en C) pour garantir la terminaison des programmes. L'élimination des constructions superflues et sujettes à erreur peut opposer le langage dédié aux langages dits Turing-complets en référence à la machine de Turing [Tur37]. Un langage Turing-complet, Turing-équivalent ou universel a un pouvoir de calcul équivalent à une machine de Turing universelle. Or, selon la thèse de Church-Turing [BA05], une machine de Turing universelle peut effectuer n'importe quel calcul qu'un équipement informatique peut effectuer. Néanmoins, le raccourci consistant à dire qu'un langage dédié ne peut être Turing-complet est erroné, la spécificité d'un langage et sa Turing-complétude sont deux notions orthogonales. Certains langages dédiés tels SQL [KT03], LaTeX [MGB+04] sont Turing-complets.

Cependant, la thèse de Church-Turing ne fait pas état des efforts à consentir pour écrire des programmes sur de telles machines. Il est certain qu'il est possible de résoudre le problème mais le programme peut devenir très complexe à écrire. Un langage dédié vise à réduire cet effort en simplifiant les programmes à écrire. Par conséquent, du fait des notations et des abstractions spécifiques au domaine, certains langages dédiés sont dits déclaratifs dans le sens où le programmeur décrit le *quoi faire* plutôt que le *comment faire*. Le langage dédié est alors considéré comme un langage de spécification cachant les détails d'implémentation au programmeur. Un outil logiciel se charge par la suite de la traduction vers une représentation plus proche de la machine hôte.

L'outil Yacc [Joh75] et le langage associé du même nom illustrent parfaitement ces trois précédents points, abstractions et notations spécifiques, restrictions et langage déclaratif. Yacc est un outil permettant à partir d'une spécification de grammaire non-contextuelle de générer un analyseur syntaxique reconnaissant tout programme respectant cette grammaire. La spécification de la grammaire est très proche de la grammaire originelle du langage formalisée à l'aide de la méta-syntaxe BNF. Au delà de l'analyse syntaxique pure du programme en entrée, Yacc permet, à partir de ces entrées, d'effectuer des calculs. Pour cela, des pseudo-variables permettent d'avoir accès aux éléments reconnus pour définir des actions sémantiques. La génération de l'analyseur syntaxique s'effectue automatiquement. Cet analyseur syntaxique est écrit en langage C et implémente un algorithme complexe [AJ74] fondé sur un automate à états finis et une pile.

Dans le reste du chapitre, nous décrivons le processus de réalisation d'un langage dédié. Néanmoins, ce processus est long et complexe et il faut bien en mesurer les tenants et les aboutissants.

3.2 Quand et pourquoi

Les moments opportuns à la réalisation d'un langage dédié sont principalement de deux natures. Premièrement, quand l'expérience montre au programmeur qu'il n'écrit que des programmes très semblables dans leur structuration, leurs abstractions et leurs finalités. Quand, leurs points communs surclassent leurs variations les uns par rapport aux autres, on dit que ces programmes font partie de la même famille de programmes [Par76]. Deuxièmement, quand le développement de programmes demandent une expertise programmatique orthogonale à celle du domaine. En effet, deux dimensions entrent en jeu dans la réalisation d'un langage dédié, le domaine cible et les utilisateurs. Le langage dédié doit permettre de résoudre les problèmes du domaine tout en respectant l'expertise des utilisateurs, potentiellement en abstrayant des complexités non-dépendantes du domaine considéré. La perspective d'un langage réellement adapté et spécialisé dans le domaine cible est attirante. Néan-

moins, il faut bien mesurer les problèmes potentiels et les avantages éventuels d'une telle approche.

3.2.1 Problèmes potentiels

La réalisation de langage dédiés souffre d'un problème endémique, le coût de développement ; d'un écueil important, la définition du domaine et sa relation avec des utilisateurs potentiels du langages ; de croyances anciennes partiellement fondées, leur manque d'efficacité, les difficultés à les maintenir et les faire évoluer.

3.2.1.1 Coûts

L'inconvénient principal dans la réalisation d'un langage dédié est le coût. Il existe deux composantes principales, le coût de développement et le coût de formation. D'une part, développer un nouveau langage demande toujours un effort considérable d'autant plus qu'il est dédié à un domaine. Le développeur des outils logiciels de support à ce langage doit appréhender les notions du domaine tout en les intégrant dans son expertise en génie logiciel.

D'autre part, malgré le fait que le langage soit dédié, il demeure un grand bouleversement dans les habitudes des utilisateurs et demande une phase d'adaptation. En effet, nous ne pouvons pas considérer le langage seul [Wil04]. L'introduction d'un nouveau langage implique l'introduction de nouveaux programmes (artefacts) écrits dans ce langage. Pour traiter ce programme, il faut de nouveaux outils comme un générateur ou un environnement de développement intégré. Ces outils produisent du code qui vient s'insérer dans une nouvelle infrastructure. De plus, la complexité de l'ensemble du langage, des programmes et des outils et de l'infrastructure vient modifier la méthodologie de travail. On parle d'innovations intrusives qui nécessitent de former les utilisateurs. Ils doivent se sentir confortables avec chacune de ces innovations avant de pouvoir être à nouveau aussi efficaces dans la résolution de problèmes du domaine. L'objectif est, qu'à terme, ils deviennent plus efficaces.

Aucune étude n'existe à l'heure actuelle sur l'amortissement de l'investissement du développement de tels langages par rapport aux gains de productivité et à l'utilisation de langages généralistes et de librairies de fonction. Nous pensons même qu'une étude de ce type soit impossible à réaliser. En effet, pour mesurer leur rentabilité, il faudrait avoir des points de comparaison pour mettre en perspective les coûts et les gains. Cela nécessite le développement en parallèle de plusieurs implémentations équivalentes. De plus, il faudrait que les développements soient effectués par des personnes présentant le même niveau d'expertise.

3.2.1.2 Domaine et utilisateurs

Le deuxième inconvénient dans la réalisation d'un langage dédié est l'adéquation nécessaire entre des utilisateurs et un domaine d'application. On peut considérer aussi que les utilisateurs font partie intégrante du domaine auquel un langage est dédié. Néanmoins, ces utilisateurs et leurs habitudes sont souvent délaissés lors de la conception d'un langage dédié. Les langages dédiés ainsi conçus ne répondent plus aux problématiques de la communauté d'utilisateurs visée et deviennent inutilisables. Cela a pour conséquence que des langages potentiellement moins intelligents, dans le sens où peu de vérifications sont effectuées, mais centrés sur les attentes des utilisateurs deviennent des standards *de facto*. Nous pensons notamment au langage PHP [LTM06] qui est largement utilisé dans la conception de pages internet dynamiques et le langage BigWig [BMS02], issu de la communauté académique présentant des abstractions dédiées telles que les sessions dont l'utilisation reste anecdotique.

Une conséquence du manque de considération de la définition du domaine (utilisateurs compris) est l'effet *Tour de Babel*. L'effet *Tour de Babel* se caractérise par une profusion de langages différents

dédiés à un même domaine donc présentant une intersection mais néanmoins divergeant sur certains points. Cette profusion de langages nuit à l'émergence d'un réel standard sur lequel capitaliser et servant de fondation à de nouveaux outils.

3.2.1.3 Anciennes croyances

Deux anciennes croyances largement répandues nuisent à l'adoption des langages dédiés. Un langage dédié est souvent considéré comme moins efficace qu'un langage généraliste par essence. Cette perte d'efficacité n'est cependant pas toujours significative et parfois même inexistante comme le montre différentes études [RM01, TCM98, MHD+07]. Certains langages dédiés, tels Devil [MRC+00], se compilent vers des langages de bas niveau comme C et ne souffrent d'aucune pénalité de performance. Cette croyance provient de la nécessité pour certains langages dédiés d'un support d'exécution (infrastructure) relativement conséquent. Néanmoins, cette perte de performance n'est pas seulement réservée aux langages dédiés. Tous les langages de haut niveau (offrant des abstractions de haut niveau) sont concernés.

Le développeur d'un langage dédié est généralement une personne différente de l'utilisateur. Dans le cas général, un utilisateur du langage n'a pas l'expertise nécessaire à l'évolution et à la maintenance des outils de support au langage, notamment le compilateur. Les langages dédiés sont donc considérés comme peu évolutifs et difficilement maintenables au sein d'une organisation potentiellement dépourvue de l'expertise nécessaire [vDK97]. Néanmoins, deux contre-arguments s'opposent à cet état de fait. D'une part, il est souvent bien plus facile de faire évoluer un compilateur de langage dédié qui est bien plus réduit qu'un langage généraliste. D'autre part, certains outils de haut niveau [dGSA07], apparaissent pour limiter les notions de compilation, à proprement parler, nécessaires au développement de langages dédiés. Nous venons de voir les problèmes potentiels qui peuvent survenir durant le développement d'un nouveau langage dédié. Néanmoins, nous allons montrer que l'introduction d'un langage dédié présente plusieurs avantages par rapport à l'utilisation d'un langage généraliste.

3.2.2 Bénéfices escomptés

La raison première du développement d'un langage dédié est un objectif de productivité. La productivité est définie, en économie, comme le rapport entre la production d'un bien ou d'un service et l'ensemble des entrées nécessaires pour le produire. Elle constitue une mesure de l'efficacité avec laquelle une économie met à profit les ressources dont elle dispose pour fabriquer des biens ou offrir des services. Les trois premiers bénéfices, facilité de programmation, sûreté et réutilisabilité, sont à mettre en perspective directe avec le gain en productivité par rapport à la ressource qu'est l'utilisateur d'un langage dédié. L'optimisabilité, les opportunités d'optimisation, est un autre bénéfice potentiel, permettant d'augmenter la productivité mais cette fois en réduisant les ressources matérielles nécessaires comme le processeur ou la mémoire utilisée. Cette optimisabilité définit la facilité à optimiser les ressources matérielles nécessaire à l'exécution d'un programme.

3.2.2.1 Facilité de programmation

Une des caractéristiques premières d'un langage dédié est de fournir des abstractions et des notations spécifiques à un domaine.

Ces éléments du langage, associés à une certaine structuration du programme, rendent le langage accessible aux utilisateurs du domaine au niveau syntaxique et au niveau sémantique. La syntaxe du programme leur paraît naturelle par l'exposition du *quoi faire* plutôt que le *comment faire*. La sémantique des abstractions et notations est également clairement établie.

Nous revenons à l'exemple de Yacc pour illustrer la facilité de programmation d'un langage dédié en trois points : naturel de la syntaxe, déclarativité du langage et concision. Premièrement, Yacc, nous l'avons vu, génère depuis la grammaire d'un langage de programmation l'analyseur syntaxique associé à cette grammaire. Le langage associé à Yacc est très proche de BNF qui est le standard *de facto* pour écrire des grammaires de langage. La première étape de la conception d'un langage consiste à définir sa syntaxe en la couchant sur papier par l'entremise de la metasyntaxe BNF. Ainsi, l'écriture d'une grammaire en Yacc consiste seulement à recopier cette première grammaire sur papier en la modifiant légèrement. Cette proximité entre la BNF et le langage associé à Yacc permet d'éviter la traduction potentiellement source d'erreurs de la spécification papier dans un langage différent. Deuxièmement, la spécification en entrée de l'outil Yacc ne décrit pas l'algorithme utilisé par l'analyseur syntaxique mais seulement la grammaire reconnue. L'algorithme, assez complexe, implémenté par le code généré par l'outil Yacc est très éloigné de cette grammaire. Il consiste en un automate à états qui nécessite de larges tables de transitions. L'implémentation de ces tables de transitions en C se fait par l'intermédiaire de larges tableaux d'entiers. Par conséquent, la spécification de la grammaire en Yacc est bien plus concise que le code généré. À titre d'exemple, nous prenons la spécification de la grammaire du format *Web Server Logs* du serveur internet d'apache. Ce format est utilisé pour enregistrer diverses informations concernant les requêtes reçues par un serveur. La grammaire représente 185 lignes de Yacc pour 1650 lignes de C générées.

La facilité de programmation accroît la productivité mais il faut également garantir la sûreté de fonctionnement des programmes développés.

3.2.2.2 Sûreté

Par sûreté d'un programme, nous entendons que l'exécution de ce programme ne peut compromettre la plate-forme d'exécution. Par conséquent, pour garantir la sûreté d'un programme, on cherche à détecter toutes les erreurs d'exécution possibles.

Dans un langage généraliste, la sûreté d'un programme n'est pas assurée. Le seul moyen pour un langage généraliste de détecter les erreurs d'exécution est par vérification. Or, un langage généraliste n'est pas vérifiable par construction. D'une part, un langage généraliste doit permettre de tout programmer. Il est donc difficile de définir des propriétés qui peuvent être vérifiées dans le contexte général. D'autre part, certaines vérifications ne sont même pas possibles car certaines propriétés d'un programme ne sont pas décidables.

Dans le cas de C, par exemple, peu de vérifications sont réellement effectuées. En C, il est possible de compromettre gravement la plate-forme sous-jacente par des erreurs de programmation (*e.g.* en réservant de la mémoire indéfiniment sans la libérer). Ce peu de vérifications s'explique par deux raisons. La première est qu'étant un langage généraliste C doit permettre de tout programmer. La deuxième est que certaines vérifications ne sont pas possibles, certaines propriétés n'étant pas décidables à cause de l'expressivité des langages généralistes.

Les langages dédiés, eux, garantissent cette sûreté. Une première partie de cette garantie est obtenue par construction, la seconde par vérification.

Un langage dédié peut être restreint dans sa syntaxe et sa sémantique pour rendre décidables certaines propriétés critiques du domaine. Ainsi, il est impossible d'écrire du code malveillant car les menaces potentielles ont été déduites du langage. Prenons l'exemple d'un langage permettant d'écrire des services de téléphonie pour illustrer les bénéfices de cette approche en termes de sûreté de fonctionnement. Un service de téléphonie est une politique de routage d'appels attachée à un utilisateur. Cet utilisateur définit ses propres critères de routage d'appel. Le langage dédié *Session Processing Language* (SPL) [BCL+06b, BCL+06a, PCRL07] permet de définir des services de téléphonie sûrs

sur une infrastructure basée sur le protocole SIP. La conception du langage SPL garantit par construction qu'un appel ne peut être perdu, que tout appel recevra une réponse, et qu'un appel ne peut plus être redirigé du moment où quelqu'un a décroché pour répondre. Les analyses de programme sont facilités par l'introduction d'abstractions spécifiques. Ces analyses sémantiques permettent de détecter des anomalies subtiles liées au domaine. Nous pouvons encore une fois revenir à Yacc qui vérifie statiquement les conflits dans la spécification. Il y a conflit dans une grammaire lorsque deux règles différentes peuvent s'appliquer en même temps. La grammaire est alors dite ambiguë puisque l'analyseur syntaxique ne peut choisir la règle à appliquer. Yacc détecte également les définitions cycliques. On parle de définition cyclique quand une règle fait référence à elle-même (par un moyen direct ou indirect).

Bien qu'augmenter la sûreté des programmes permette d'augmenter la productivité, d'autres facteurs peuvent également y contribuer. La systématisation de différentes formes de réutilisabilité en est un.

3.2.2.3 Réutilisabilité

Un langage dédié est conçu pour guider l'utilisateur vers une réutilisation systématique. D'une part, un langage dédié intègre des connaissances et une expertise liées à un domaine permettant la réutilisation systématique du savoir et du savoir-faire. D'autre part, d'un point de vue pragmatique, la réutilisation revêt deux formes, la réutilisation de code et la réutilisation de schémas.

Un langage dédié garantit une réutilisation systématique du code par le biais de ses abstractions et ses opérations, qu'elles soient explicites ou implicites. Cette factorisation du code par ces éléments du langage est garantie car totalement prise en charge par le support d'exécution du langage dédié [CM98, Con04]. La plupart des environnements de programmation inclut la capacité d'abstraire des opérations communes dans des bibliothèques. Cependant, la réutilisation de ces bibliothèques est laissée à la charge et/ou à la discrétion de l'utilisateur. Par conséquent, la réutilisation est possible mais pas automatique. L'autre forme de réutilisation est la réutilisation de schémas. Un langage dédié permet d'abstraire des schémas d'opérations courants qui ne sont pas exprimables dans des langages généralistes. Par exemple, le langage SPL structure le programme sous la forme de gestionnaires d'événements, chacun associé à un type de message SIP. La décomposition du problème initial en sous-problèmes élémentaires est directement offerte par la structure du message.

Le dernier point permettant d'améliorer le gain de productivité induit par l'utilisation d'un langage dédié est l'optimisabilité inhérente à une solution spécifique et de haut niveau.

3.2.2.4 Optimisabilité

Les langages dédiés sont plus spécifiques et de plus haut niveau que les langages généralistes. La spécificité d'un langage dédié permet des optimisations spécifiques inaccessibles à un langage généraliste implémentant la même tâche. Il est possible d'optimiser les constructions abstraites. Ainsi, dans le langage Teapot [CRL96], qui permet d'implémenter des protocoles de cohérence de mémoire cache, certaines constructions sont optimisées. Notamment, la primitive `Suspend`, permettant de suspendre un traitement, est optimisée pour réduire l'état à sauvegarder et à restaurer. Un langage dédié, de par son haut niveau, permet de raisonner plus facilement sur l'ensemble du programme, en profitant d'une vision plus globale. Le compilateur SQL[KT03] traduit une requête dans un modèle de requête fondé sur un graphe. Ensuite, la partie optimisation du compilateur génère de nombreux plans d'exécution différents pour satisfaire la requête. Pour estimer le coût de chacune de ces alternatives, le compilateur examine différentes statistiques. Le plan d'exécution le plus intéressant est sélectionné.

Ensuite, un plan d'accès aux différentes tables est mis en place et optimisé grâce à divers algorithmes. Cette optimisation est rendue possible par la manipulation par le compilateur SQL des concepts clés du domaine.

Nous l'avons vu, la réalisation d'un DSL n'est pas un processus facile car les risques encourus en cas d'échec sont nombreux tout comme le sont les bénéfices potentiels. Ainsi, un langage dédié peut permettre d'augmenter la productivité [KMB+96]. Par exemple, une étude [Wei96] a montré que l'introduction de langages dédiés à Bell Labs a entraîné un accroissement de la productivité par un facteur compris entre quatre et cinq. Néanmoins, pour tirer pleinement parti de tous ces bénéfices et ainsi maximiser la gain en productivité, un langage dédié doit être conçu de manière méthodique et systématique.

3.3 Quoi

La conception d'un langage dédié devrait en théorie s'appuyer sur une analyse de domaine et une analyse de famille de programmes. Une analyse de domaine consiste à étudier un domaine et à en déterminer les éléments d'information réutilisables. L'analyse de famille de programmes est l'analyse d'un ensemble de programmes apparentés par de nombreux points communs. Les résultats de ces analyses combinées forment les entrées du processus de conception en lui-même.

3.3.1 Analyses préliminaires

La notion d'analyse de domaine est apparue avec la recherche de la réutilisabilité des composants logiciels. Or, un moyen de garantir une certaine réutilisabilité est l'utilisation d'un langage dédié. Ainsi, l'analyse de domaine a été détournée pour être utilisée dans la conception de langages dédiés. Elle consiste en une analyse de plus haut niveau que l'analyse de familles. En effet, l'analyse de domaine d'après la définition donnée par Jim Neighbors *tente d'identifier les objets, les opérations et les relations entre ce que les experts du domaine considèrent comme important pour ce domaine* [Nei80]. L'analyse de famille, quant à elle, s'applique à l'implémentation de programmes réels.

Le but premier de l'analyse de domaine est la définition de ce domaine pour identifier et structurer les informations réutilisables. Pour cela, elle doit s'appuyer sur de nombreuses sources de documentation, des exemples et des contre-exemples. Néanmoins, il n'existe pas de moyen formel de définir un domaine et d'après Prieto-Diaz [PD90], un domaine est en fait un réseau semi-hiérarchisé de domaines plus petits et différents. Malgré diverses méthodologies proposées [McC85, Ara89], les analyses de domaine sont souvent effectuées de manière *ad hoc* ce qui conduit généralement à leur échec.

Un point noir de l'analyse d'un domaine est qu'elle ne s'intéresse qu'aux points communs qui existent entre les différents éléments d'un domaine. Or, il est également nécessaire de s'intéresser aux particularités propres à chaque élément pour identifier les variations à l'intérieur d'un domaine. De plus, l'analyse de domaine ne s'appuient pas sur des éléments concrets et fiables, par opposition à l'analyse de famille qui s'attache à l'étude de programmes fonctionnels. Cette approche permet de retirer des enseignements plus pertinents par rapport à ceux émanant de documentation ou d'entretiens avec des utilisateurs du domaine.

Réunir des programmes en familles est une idée assez ancienne qui a été introduite pour diminuer les efforts de développement et de maintenance de programmes apparentés [Par76]. Parnas ne définit pas exactement ce qu'est une famille de programmes mais donne un indice sur le moment de l'identification d'une famille de programmes. D'après lui, une famille de programmes est identifiée, lorsqu'il est utile d'étudier d'abord les propriétés communes aux différents programmes puis les particularités

de chaque membre de la famille. L'idée est de s'opposer à une implémentation exhaustive d'un programme dans son ensemble puis d'en dériver les autres programmes, on parle alors de terminaison séquentielle. Le but est de définir un socle commun partagé entre tous les programmes. Pour illustrer cette notion, nous pouvons dire comme Parnas que les différentes versions d'un même système d'exploitation constitue une famille.

Parnas propose deux méthodologies pour développer des familles de programmes. La première est fondée sur des raffinements successifs de programmes incomplets vers des programmes complets. La seconde est fondée sur l'utilisation de modules. Dans un travail plus récent [Wei96], Weiss a présenté la méthode FAST qui a été introduite avec succès à Bell Labs. FAST est un processus de génie logiciel répondant à deux problématiques, comment identifier et représenter une famille de programmes et comment générer un membre de cette famille à partir d'une spécification concise. La première problématique concernant l'identification est résolue par l'introduction de l'analyse des points communs [Wei98]. Cette analyse se décompose en trois volets, 1) définir un dictionnaire des termes généralement utilisés, 2) exprimer les hypothèses vraies pour tout membre d'une famille, 3) décrire le spectre des variations acceptables à l'intérieur d'une même famille. La deuxième problématique est résolue par l'implémentation d'un environnement de génie logiciel pour les applications dont une partie est la conception de l'équivalent d'un langage dédié.

Nous avons montré dans cette section que l'analyse de domaine n'est pas suffisante comme entrée à la conception d'un langage dédié, car elle se focalise sur les points communs en occultant des variations à l'intérieur du domaine. Nous avons ensuite mis en avant la notion de famille de programmes qui est un support fiable pour le développement de langages dédiés. L'analyse de points communs introduite par Weiss, qui est une forme d'analyse de famille, a été utilisé avec succès pour la conception de langages dédiés. Néanmoins, l'approche FAST n'a pas été conçue avec l'objectif d'être une méthodologie claire à la conception d'un langage dédié. Dans la section suivante, nous mettons l'accent sur des travaux visant à exprimer un processus systématique de conception de langage dédié.

3.3.2 Processus de conception

La méthodologie Sprint [CM98], contrairement à l'approche FAST, est une première méthodologie visant le développement de langage dédié. Cette méthodologie réconcilie les différentes visions possibles d'un langage dédié, à savoir une vision langage et une vision architecture logicielle. Pour cela les auteurs utilisent une méthodologie formelle pour développer des langages génériques, tout en exprimant des préoccupations d'architecture logicielle. La méthodologie définit les sept étapes chronologiques qui interviennent dans le développement d'un langage dédié. En premier lieu, une analyse de langage est effectuée. Cette analyse de langage est l'équivalent d'une analyse de points communs. Les résultats de cette analyse sont des objets et des opérations nécessaires pour exprimer les solutions aux problèmes considérés, des obligations que doit remplir le futur langage. Ils servent à définir la syntaxe du langage dédié ainsi qu'une sémantique informelle. Cette sémantique est divisée en une sémantique *statique* et une sémantique *dynamique* à l'instar de celle d'un langage généraliste. Cette séparation rend les étapes de configuration explicite, par égard aux préoccupations d'architecture logicielle. La sémantique du langage est ensuite formellement définie par les fonctions de valuation des constructions syntaxiques. Une fois la sémantique formellement définie, les éléments de la sémantique dynamique sont regroupés pour former une machine abstraite dédiée. L'implémentation des fonctions de valuation sous forme d'interprète repose sur l'implémentation de la machine abstraite. Les fonctions de valuation peuvent également être implémentées sous la forme d'un compilateur vers les instructions de la machine abstraite. L'étape finale est une phase d'optimisation fondée sur un évaluateur partiel [CLLM04] pour combiner la flexibilité de l'interprétation avec les performances de la

compilation.

Cette méthodologie a été récemment revisitée [Con04] à la lumière des langages dédiés développés entre temps par les auteurs. La nouvelle méthodologie est une approche beaucoup plus pragmatique que la précédente car centrée sur la notion de famille de programmes. Elle part de l'observation que l'identification d'une famille de programmes donne naturellement naissance à une librairie de fonctions. La nouvelle méthodologie s'attache ensuite à identifier et décrire les étapes systématiques du passage d'une librairie à un langage dédié. Elle propose également l'utilisation de la famille de programmes étudiée comme point de comparaison pour l'évaluation du langage nouvellement conçu.

Dans les précédentes sections, nous avons défini les moments opportuns et les motivations à la conception d'un langage dédié. La conception d'un langage dédié est une étape complexe ne serait-ce que dans la délimitation du domaine d'application. Néanmoins, une fois le langage conçu, il reste à implémenter le support logiciel de ce langage. Dans la section suivante, nous nous attachons à présenter le *comment* implémenter un tel langage.

3.4 Comment

Nous avons vu précédemment qu'un langage dédié est restreint, nous avons parlé de *petit* langage. Néanmoins, il serait faux de penser que l'implémentation d'un langage dédiée est elle aussi *petite*. En effet, un langage dédié est généralement haut niveau ; le fossé à combler par la compilation du langage s'avère important. Cette thèse ne traite pas des techniques d'implémentation d'un langage dédié. Cependant, dans cette section, nous présentons quelques travaux antérieurs significatifs utilisables dans ce contexte. Cette section n'a donc pas vocation à être exhaustive.

Pour clarifier la présentation, nous opérons une séparation entre les langages dédiés autonomes et les langages dédiés enchâssés. Un langage est dit enchâssé lorsqu'il est implémenté comme une extension d'un langage généraliste.

3.4.1 Langages autonomes

Un langage autonome est un langage indépendant de tout autre langage préalablement développé. L'inconvénient de cette approche est qu'un langage autonome doit être développé entièrement, à partir de zéro, sans aucun appui sur un langage faisant office de fondation. Cependant, cette approche permet de contrôler entièrement le langage sans contrainte préalable. Ici, nous faisons la distinction entre les techniques d'implémentation de compilateurs et d'interprètes de langages autonomes.

Un compilateur va générer du code fondé sur des instructions de la machine abstraite. L'interprète va, quant à lui, prendre en charge l'exécution en faisant appel aux fonctionnalités fournies par la machine abstraite. Par un raccourci, nous pouvons dire que la machine abstraite représente alors le contexte d'exécution du programme.

3.4.1.1 Implémentation de compilateurs de langage dédié

L'approche classique pour implémenter un compilateur d'un langage dédié est d'utiliser les outils historiques de construction de compilateurs. Les plus connus sont Lex [LS75] et Yacc [Joh75] qui génèrent automatiquement des analyseurs lexicaux et syntaxiques, respectivement. Hormis ces derniers, même si quelques travaux ciblant différentes phases de la compilation existent [GHL+92, PDC92, Sch97], le reste de l'implémentation du compilateur se fait généralement manuellement à l'aide d'un langage généraliste.

Les concepteurs de l'approche DRACO [Fre87] cherchaient à améliorer la réutilisabilité des composants logiciels qu'ils développaient. Leur solution passe par l'utilisation d'un langage dédié. L'approche consiste en effet à spécifier la syntaxe d'un langage et un ensemble de transformations de programme. La spécification de la syntaxe est utilisé pour générer un analyseur syntaxique alors que les transformations de programmes sont utilisés pour raffiner les éléments du langage dédié dans des éléments d'un système exécutable. Plus récemment, l'approche Stratego/XT [BTKVV06] a été développé dans le même but. Stratego [Kal06] est un langage de transformation d'arbre de syntaxe abstraite, de programmes définis à l'aide de termes. XT apporte d'autres outils de transformation pour l'analyse syntaxique et la mise en forme de code. Le désavantage de telles approches est que pour profiter pleinement de l'apport des outils, il est nécessaire que tous les éléments du compilateur reposent sur ces outils.

Le développement d'un compilateur pour un langage dédié peut reposer sur des outils utilisés dans le contexte des langages généralistes. Néanmoins, la nature des langages dédiés rend leur compilation très différente de la compilation de langages généralistes communs. En fait, généralement, le compilateur d'un langage dédié produit du code écrit dans un langage généraliste ; les aspects bas-niveau de la compilation étant laissés au compilateur du-dit langage généraliste. Le compilateur d'un langage dédié projette les informations haut niveau et les caractéristiques du domaine vers le langage généraliste ciblé et les couches sous-jacentes (*e.g.*, intergiciel et protocoles). Le code ainsi généré reprend des patrons de code, des motifs qui se répètent, qui s'avèrent assez importants [Cle88]. La génération de ces patrons peut être un processus complexe reposant sur diverses conditions. Sans support d'outils spécifiques, le processus de génération devient fastidieux et sujet à erreur. De plus, le traitement de ces patrons occultent les préoccupations spécifiques aux domaines exposés par un programme. Les auteurs de [CLRC05] proposent une méthodologie novatrice s'appuyant sur des outils de génération de programmes tels que les aspects [Kic96] et les annotations [CEI+07]. La méthodologie se décompose en deux phases, la compilation de la logique du programme puis l'utilisation des outils de génération de programmes. Dans un premier temps, la logique du programme écrit dans un langage dédié est compilée vers une représentation abstraite dans un langage généraliste. Cette représentation est abstraite car elle inclut des opérations dont l'interprétation n'est pas encore définie. Ensuite, les outils de génération de programmes sont utilisés pour définir la compilation de différentes facettes spécifiques au domaine considéré. Une catégorie de facettes établit la correspondance d'un programme dans un langage dédié dans l'environnement d'exécution ciblé. Une seconde catégorie est dévolue à la compilation des abstractions du langage dédié. Une troisième catégorie définit la génération de code associée au programme considéré. Cette approche permet de modulariser le processus de génération du compilateur d'un langage dédié.

Cette dernière approche représentait les dernières avancées en termes d'implémentation d'un compilateur pour un langage dédié autonome. L'autre possibilité pour implémenter un langage dédié est la définition d'un interprète. Si le compilateur se montre plus efficace, l'interprète s'avère plus facile à implémenter. L'interprète traite directement chaque construction du langage pour produire les résultats attendus ; le compilateur produit un programme, qui lorsqu'il est exécuté, produit les résultats. Cette indirection consistant à passer par un programme intermédiaire rend la construction d'un compilateur plus délicate. Par conséquent certains travaux se sont concentrés à rendre les interprètes plus efficaces.

3.4.1.2 Implémentation d'interprètes

Dans sa thèse [Thi98], Scott Thibault établit aussi qu'il est plus naturel d'implémenter un langage dédié en divisant l'interprète en deux niveaux : une machine abstraite dédiée à un domaine et une

couche d'interprétation. Cette division reflète les concepts sous-jacents de points communs et variations que le langage dédié appréhende. D'une part, la machine abstraite en définissant un modèle de calcul qui inclut tous les programmes de la famille de programmes, représente les points communs. D'autre part, la couche d'interprétation en établissant la correspondance entre un programme écrit dans un langage dédié et les opérations de la machine abstraite représente les variations. Néanmoins, l'inefficacité chronique dont sont victimes les interprètes est bien souvent rédhibitoire. Par conséquent, Scott Thibault propose d'utiliser l'évaluation partielle comme technique d'optimisation. Il part du constat qu'un interprète est le candidat idéal à l'évaluation partielle. En effet, un interprète est un programme *générique* au niveau de la famille de programmes considérée. Suivant le programme en entrée (écrit dans le langage dédié considéré), l'interprète effectue le traitement associé dans la famille de programmes. L'évaluation partielle, quant à elle, est une transformation de programme automatique qui adapte un programme générique à un contexte spécifique dans le but d'éliminer l'inefficacité introduite par la généricité.

Néanmoins, cette thèse ne répond pas à une problématique majeure de l'implémentation d'un langage dédié, à savoir le coût. Scott Thibault ne s'est pas focalisé sur la réutilisabilité des éléments développés. Rendre réutilisable certains éléments du langage permet de réduire les coûts en ventilant ces coûts sur plusieurs langages dédiés. Une solution est la modularisation du langage dédié. Le langage est décomposé en éléments réduits réutilisables et composables appelés modules. Une première approche exposée par Peter Mosses [Mos04] consiste à définir la sémantique du langage dédié à l'aide d'un formalisme dédié à la modularisation. L'interprète est ensuite construit par la composition de modules développés à l'aide du meta-environnement ASF+SDF [vdBHKO02]. L'idée développée par Tim Sheard [SBP99] est quelque peu différente, elle consiste à profiter des méthodes d'abstraction telles que les monades [Wad90] et l'étagement [TBS98] de l'interprète pour partitionner chaque décision de conception dans un module séparé. La sémantique du langage est définie à l'aide de la sémantique dénotationnelle [Sch86] puis par la capture des effets du langage d'implémentation et de l'environnement d'exécution dans une monade, l'interprète est réécrit dans un style monadique. Le nouvel interprète est ensuite étagé à l'aide de techniques de meta-programmation.

L'inconvénient d'un langage dédié autonome outre la difficulté potentielle à l'implémenter est la difficulté à le faire adopter dans le contexte de programmeurs expérimentés avec des préférences particulières. En effet, qui dit langage dédié dit syntaxe dédié et bouleversement des habitudes des utilisateurs. Or, généralement, un programmeur a une sensibilité pour un paradigme particulier voire pour un langage particulier ; rendre le langage dédié à implémenter proche de ces considérations facilite son adoption. De plus, il est parfois souhaitable de pouvoir spécialiser un langage généraliste dans des contextes particuliers, par exemple l'accès à une base de données. Le langage généraliste est utilisé pour les tâches courantes mais un langage réduit embarqué permet de spécifier des tâches très spécifiques. Enfin, fonder le développement d'un langage dédié sur un langage existant éprouvé réduit les efforts nécessaires. Pour ces raisons, les approches visant à implémenter des langages enchâssés sont plébiscitées.

3.4.2 Langages enchâssés

Paul Hudak dans son article fondateur [Hud96] part du constat que personne ne veut construire un langage de programmation à partir de zéro. Il propose d'hériter l'infrastructure d'un autre langage et de l'adapter de certaines manières au domaine d'intérêt. Il insiste notamment sur le fait que son équipe a implémenté de nombreux langages enchâssés dans le contexte du langage Haskell [HF92]. L'idée est encore une fois d'utiliser les monades pour ajouter des fonctionnalités et des optimisations dédiées. L'interprète peut alors être vu comme une succession de cercles concentriques, chaque cercle

représentant une nouvelle fonctionnalité du langage. À chaque niveau, de nouvelles fonctionnalités peuvent être ajoutées, avec leur sémantique, sans altérer aucun des niveaux précédents. Bien que séduisante, cette approche ne fut que marginalement adoptée pour diverses raisons. Haskell n'est pas un langage largement répandu et l'utilisation de monades est considérée comme complexe. De plus, Hudak propose une approche mais aucun outil de support à cette approche.

Récemment, de nouvelles approches et outils sont apparus visant à enchâsser des langages dédiés dans des langages généralistes *grand public*. Ces outils sont venus dans le sillage d'outils populaires comme ANTLR [PQ95] ou JavaCC [Jav] qui ont le même but que Yacc et qui fournissent avec leur distribution la grammaire du langage Java [GJS96]. Notamment, JastAdd [EH07b] est un système de compilateur de compilateurs fondé sur Java. Ce système repose sur un langage du même nom [EH07a] qui étend Java par le support d'un formalisme de spécification appelé ReCRAGs pour *Rewritable Circular Reference Attributed Grammars*. Ce formalisme permet d'élever le niveau d'abstraction pour les traitements fondés sur les grammaires en utilisant des techniques déclaratives pour réécrire des arbres de syntaxe abstraite. Le système JastAdd a été utilisé par exemple pour implémenter de manière modulaire un compilateur Java version 1.5 depuis un compilateur Java version 1.4. Dans le même esprit, Silver [VWS07] est un langage de spécification de grammaires attribuées extensible. Ce langage a notamment été utilisé pour implémenter une version extensible de Java 1.4 [VWKSB07] et divers modules d'extension. Par exemple, un module permet d'enchâsser SQL dans Java et d'effectuer des vérifications de type sur les requêtes SQL [VWBH06].

Une approche différente car plus fixée sur les objectifs que les moyens est xtc[1] pour eXtensible C. Xtc qui est un système fondé sur le concept de *macros* permettant de développer des extensions au langage C. L'objectif est d'étendre le langage C pour faciliter le développement de systèmes d'exploitation et de systèmes distribués. Xtc a été utilisé dans le cadre de Jeannie [HG07] qui permet d'entremêler le langage C et le langage Java.

3.5 Bilan

Ce chapitre définit un langage dédié comme étant un langage de programmation qui est spécifique à un domaine d'application particulier. Un langage dédié permet de résoudre certains problèmes de manière plus simple et plus sûre qu'un langage généraliste ce qui améliore la productivité associée. Néanmoins, si un langage dédié présente de nombreux avantages en termes de réutilisabilité systématique notamment, sa conception doit être le fruit d'un travail rigoureux suivant une certaine méthodologie précise telle que celles que nous présentons. Une fois ce processus de conception achevé, il reste à implémenter le compilateur ou l'interprète associé au langage dédié. La dernière section de ce chapitre s'attache ainsi à la présentation de quelques techniques d'implémentation significatives pour l'implémentation de langages autonomes et de langages enchâssés.

[1] `http ://cs.nyu.edu/rgrimm/xtc/`

Chapitre 4

État de l'art

Dans le chapitre 2, nous avons décrit les communications réseaux, en insistant sur la notion de protocole. Un protocole est un ensemble de règles qui régit la communication. Les protocoles peuvent être séparés en deux catégories : les protocoles des couches basses et les protocoles des couches hautes ou couche application dans le modèle TCP/IP. Le support des protocoles des couches basses est fourni avec le système d'exploitation. Le support des protocoles des couches hautes est à la charge du programmeur de l'application. Or, le code de ce support doit relever plusieurs défis, la robustesse, l'efficacité et la maintenabilité. Les langages dédiés, décrits dans le chapitre 3, apportent de nouvelles solutions à certains problèmes de développement logiciel. Ils permettent, s'ils sont bien conçus, d'assurer une réutilisabilité systématique tout en garantissant une certaine sûreté sans compromission de performance.

Nous présentons dans ce chapitre divers travaux, préalables à celui exposé dans cette thèse, relatifs au support protocolaire d'applications réseau et notamment l'analyse syntaxique des messages protocolaires. Nous nous concentrons notamment sur les solutions fondées sur un langage dédié. Nous commençons néanmoins par une digression sur l'utilisation de langages avancés pour la programmation système et réseau. Par langage avancé, nous entendons un langage qui vérifie certaines propriétés comme le typage tout en automatisant certains traitements comme la gestion du stockage des données.

4.1 Langages avancés pour la programmation système et réseau

La conception de langages de programmation a largement progressé depuis les années soixante-dix. Néanmoins, la plupart des programmes système et réseau sont toujours écrits dans des langages qui existaient il y a trente ans ou qui intègrent peu des caractéristiques de programmation avancée. Aujourd'hui, les programmes système et réseau sont écrits en C ou C++. Les avantages de ces langage incluent la vitesse d'exécution des programmes, la possibilité d'exprimer des opérations bas-niveau et la compatibilité avec des systèmes existants. Ces avantages sont jugés prépondérants par rapport aux inconvénients inhérents à l'utilisation de ces langages : manque de modularité, typage faible et gestion manuelle du stockage des données. Par conséquent, le défaut prépondérant de ces langages est le manque de sûreté. En effet, il n'existe pas de typage pour garantir la correspondance entre le prototype d'une fonction et les arguments qui lui sont réellement passés. De même pour la sûreté de stockage, qui garantit l'usage correct de la mémoire.

De nombreux travaux tentent d'introduire des langages avancés dans le contexte de la programmation système et réseau. Le projet FoxNet [HL94, HLP98, BRL01] est une implémentation d'une pile réseau incluant les protocoles TCP et IP. Une pile réseau offre aux couches supérieures un support

complet des protocoles sur lesquels elle est fondée, en l'occurrence TCP et IP. La structure de la pile FoxNet est fondée sur celle du X-kernel [HP91]. Dans cette architecture, chaque protocole présente essentiellement la même interface au monde extérieur. Cela permet de combiner des protocoles d'une manière transparente, par exemple en ayant une instance de TCP qui fonctionne directement au-dessus d'Ethernet. Outre cette structure innovante, la pile FoxNet est implémentée à l'aide d'une extension au langage Standard Meta Language [Kam96] (SML) qui est un langage fortement typé avec inférence de type, une gestion automatique du stockage (à la fois pour la pile et le tas contrairement à C qui ne gère que la pile) et un ramasse-miette. Tous ces aspects garantissent une certaine sûreté mais les performances de la pile FoxNet ne s'avèrent pas être aussi efficace qu'une pile écrite en C [Der]. Elle peut atteindre un débit équivalent à celui d'une pile écrite en C à condition que la charge utile à véhiculer soit de l'ordre du mégaoctet. Néanmoins, à débit égal, la pile FoxNet consomme dix fois plus de ressource processeur. Ces problèmes de lenteur et de consommation processeur proviennent de deux caractéristiques inhérentes au langage SML. D'une part, il n'existe pas de moyen efficace d'implémenter des fonctions de moins de vingt instructions (de petites fonctions utilitaires très présentes en programmation système et réseau). D'autre part, SML n'a pas été conçu pour accèder et manipuler des structures mémoire d'autres langages, ce qui est nécessaire pour un langage de programmation système.

Le projet SPIN [BSP+05] est un système d'exploitation extensible qui se veut être flexible, sûr et efficace. SPIN est conçu autour d'un ensemble restreint de services comme les processus légers, les primitives de mémoire virtuelle, les pilotes de périphériques et le mécanisme d'extension. Les autres fonctionnalités, telles que les systèmes de fichier et le support réseau, sont implémentés comme des extensions et dynamiquement installées dans le système lorsqu'il est en fonctionnement. La particularité de SPIN est qu'il est fondé sur le langage modula-3 [SSP+96], qui à l'instar de SML est fortement typé. Un travail intégré dans SPIN qui nous intéresse plus particulièrement concerne l'architecture Plexus [FB96]. L'idée directrice est de permettre aux applications d'atteindre de hautes performances même avec des protocoles personnalisés. Le support de ces protocoles dans SPIN est considéré comme une extension qui s'installe dynamiquement dans le noyau du système d'exploitation. En s'exécutant dans le noyau, ce support peut accéder à moindre coût à l'interface réseau et autres services du système d'exploitation. L'architecture Plexus permet d'intégrer de la connaissance de l'application au niveau du support protocolaire, fournissant une architecture pour développer le support de nouveaux protocoles et permettant d'implémenter des optimisations spécifiques à l'application pour les protocoles existants. Ainsi, les applications s'exécutent de manière plus rapide, en consommant moins de cycles du processeur que les implémentations monolithiques conventionnelles.

Nous avons vu dans cette première section, les avantages à choisir un langage avancé par rapport à C dans le contexte de la programmation système et réseau. Cependant, malgré les quelques dix ans écoulés depuis ces travaux, la situation a peu évolué, principalement car les programmeurs système sont habitués à programmer en C et ne veulent pas changer. La suite du chapitre se concentre sur des langages dédiés pour le développement de support protocolaire ou de logique d'application.

4.2 Approche langage au développement de logique d'applications

Dans le chapitre 2, nous avons insisté sur le support protocolaire au niveau applicatif et sur le fait que le développement de ce support soit à la charge du programmeur de l'application. Il existe cependant quelques approches tout-intégré qui permettent d'implémenter la totalité d'une application réseau. On parle alors d'approches verticales car elles sont très spécialisées, étant focalisées généralement sur un protocole particulier. Notamment, le langage dédié SPL [BCL+06a, PCRL07]

et l'environnement associée [BCL+06b] permettent de créer des services de téléphonie sur IP au dessus du protocole SIP. Ces services consistent à définir des politiques de routage de messages, et par conséquent d'appels, en fonction de divers critères tels que l'identité de l'appelant, la raison ou l'heure de son appel. L'environnement d'exécution fournit une interface haut niveau sur laquelle vient s'appuyer le langage SPL. Cet environnement fournit notamment la notion de session spécifique au protocole SIP qui permet d'avoir une gestion d'états associée au flot des messages qui transitent. Un programme SPL doit implémenter des gestionnaires d'événements particuliers remontés par l'environnement d'exécution. La structuration de ces gestionnaires reflètent une hiérarchie entre les diverses sessions en cours au niveau de l'environnement d'exécution.

Le langage SPL intègre des restrictions et des analyses statiques qui garantissent la viabilité des services écrits. Il existe deux problèmes typiques lors de l'exécution d'une politique de routage d'appels, le blocage et le lâcher d'appels. On parle de blocage lorsque le traitement d'un appel prend trop de temps par rapport à ce qu'un utilisateur attend. Ce blocage peut intervenir, par exemple, si une boucle infinie apparaît dans le code du service de téléphonie. Or, les approches classiques à base de librairie de langages généralistes [DRM04] ne font rien concernant les erreurs qui peuvent arriver du fait de l'utilisation de ces langages. Il est donc possible de développer un service qui bloque des appels. SPL n'intègre pas d'instruction de boucle non contrôlés, toutes les itérations s'appliquent sur un domaine qui est connu statiquement. Le lâcher d'appels correspond au non-traitement d'un appel entrant. Cela se produit lorsque la réception d'un message n'entraîne pas l'exécution d'une opération de signalisant qui vise à transmettre ce message ou bien à convoyer une réponse à ce message. L'interprète SPL intègre une phase d'analyses qui vérifie que quelque soit le chemin d'exécution emprunté pour le traitement d'un message, une opération de signalisation est effectuée.

Néanmoins, les approches verticales répondent à une problématique tellement particulière que le support protocolaire est généralement figé. Ainsi, toutes les applications s'appuient sur un support protocolaire cohérent dans un domaine particulier. L'inconvénient majeur de ce type d'approches est que les restrictions introduites pour garantir certaines propriétés réduisent de manière drastique la famille de programmes considérées. Par conséquent, nous nous intéressons à des approches horizontales permettant de développer le support protocolaire auquel on peut associer une logique d'application quelconque.

4.3 Approche langage au traitement des données

Le langage dédié PADS [FG05] est une solution générique au traitement de données. Il part du constat qu'il existe beaucoup de formats de données *ad hoc* qui sont semi-structurées, par exemple les journaux d'événements dans les serveurs Web. Dans ce cas, il n'existe pas d'outils pour traiter ces données. Une description PADS est une suite de déclarations de types proche de C ; ces déclarations de type indiquent comment traiter les données associées. En plus de permettre la description du format physique des données, le langage intègre un moyen d'exprimer des contraintes sémantiques sur les données (par exemple des bornes sur des valeurs entières ou des dépendances contextuelles). Ainsi, la description PADS peut faire office de documentation formelle pour le format de données considéré. À partir de cette description, un compilateur génère de nombreux outils permettant entre autres de traiter et de transformer les données en entrée.

Une évolution récente de PADS est PADS/ML [MFW+07] qui répond à la même problématique tout en différant par trois points. Premièrement, au lieu de compiler vers du C comme en PADS, PADS/ML compile vers du O'Caml [Ler97]. Cette approche simplifie les tâches de traitement des données, comme leur filtrage ou leur normalisation, en bénéficiant de constructions avancées du lan-

gage O'caml. Deuxièmement, les types en PADS/ML peuvent être paramétrés par d'autres types ce qui rend les descriptions plus concises. Enfin, une nouvelle interface générique a été créée pour permettre l'implémentation d'outils de traitements de données personnalisés, en facilitant l'extension du compilateur.

Néanmoins, ces approches ne s'avèrent pas être adaptées au développement du support protocolaire d'applications réseaux. D'une part, leur objectif principal est d'être capable de traiter des enregistrements de données de plusieurs giga-octets (PADS fut développé pour analyser les détails d'appels pour un opérateur téléphonique) ; elles n'ont donc pas été conçues pour être réactives à des petits messages à un débit important. D'autre part, le langage en lui-même n'est pas adapté à la description de protocoles textuels car très éloigné de la *metasyntaxe* ABNF. Ainsi, il est nécessaire de traduire la spécification du protocole dans une description PADS, ce qui peut s'avérer complexe. De plus, pour les protocoles textuels, certains éléments ne peuvent être traduits en types et doivent donc être traduits sous forme d'expressions régulières. Par conséquent, même la description du protocole HTTP fournie comme exemple avec le compilateur est loin d'être satisfaisante quant à sa complétude par rapport à la spécification originelle.

Il existe des solutions dédiés au développement de support protocolaire que ce soit des protocoles binaires ou des protocoles textuels. Ces solutions sont adaptées dans leur format notamment aux besoins de la description de la syntaxe des messages d'un protocole.

4.4 Approche langage au développement du support protocolaire pour des applications réseau

Un projet précurseur concernant le support protocolaire est PacketTypes [MC00]. L'idée directrice est d'utiliser les types pour éliminer l'écriture de code bas niveau en C pour l'analyse de messages binaires bruts reçus. Or, les types dans les langages de programmation tels que C ne sont pas adaptés à la description de formats de paquets car pas assez bas niveau. Par conséquent, PacketTypes a été conçu comme un langage dédié de spécification de paquets et sert de système de types pour les formats des paquets. PacketTypes permet d'exprimer certaines caractéristiques des formats de protocoles comme l'encapsulation, des champs de taille variable ou optionnels. Néanmoins, le code généré par le compilateur s'avère 40% plus lent que du code équivalent écrit à la main ce qui est rédhibitoire pour une large adoption.

Binpac [PPSP06] est développé dans le but d'être l'équivalent de Yacc pour les protocoles applicatifs qu'ils soient binaires ou textuels. Binpac fait partie du détecteur d'intrusions réseau appelé Bro [Pax99]. Un détecteur d'intrusions est un programme qui se fonde sur des analyseurs syntaxiques et un ensemble de règles de contraintes pour définir si un message reçu est malveillant. Au delà de la seule analyse syntaxique, Binpac gère l'état des communications et différentes problématiques inhérentes aux protocoles applicatifs. Ce langage dédié est lui aussi fondé sur la description de types. Par conséquent, la phase de traduction des spécifications préalable à l'utilisation de Binpac compromet la robustesse des analyseurs syntaxiques générés. En effet, il est nécessaire de décrire certaines parties du message à l'aide d'expressions régulières. La description de l'analyseur syntaxique du protocole HTTP fournie avec la distribution de Binpac s'avère d'ailleurs incomplète ne gérant pas les en-têtes multi-lignes, notamment. Les analyseurs syntaxiques générés en C++ sont plus efficaces que les analyseurs syntaxiques écrits à la main, préalablement utilisés dans Bro. Pourtant, l'évaluation est partiale car il n'y a pas de comparaison avec d'autres implémentations. De plus, l'implémentation courante est très fortement couplée à l'implémentation de Bro. Les analyseurs syntaxiques générés lèvent des événements spécifiques au niveau supérieur de l'analyse effectuée par Bro.

Le projet GAPA et le langage associé GAPAL [BBW+07] a la même finalité que Binpac concernant la génération d'analyseurs de protocoles applicatifs. Toutefois, il existe quelques points de divergence entre ces deux approches. Binpac et GAPAL ne sont pas exactement équivalents en termes d'expressivité. En effet, GAPAL permet de définir une logique associée au protocole ce que ne permet pas Binpac. Cette logique est comparable au niveau supérieur d'analyse dans Bro. D'un premier abord, la solution GAPA+GAPAL semble donc moins modulaire que Binpac. De plus, le compilateur de Binpac permet de générer des analyseurs syntaxiques en C++, ce que ne permet pas GAPAL. GAPAL est un langage interprété extrêmement couplé au système GAPA et nécessitant l'infrastructure fournie par GAPA comme pré-requis à son exécution. Par exemple, plusieurs spécifications GAPAL sont chargées en même temps à l'intérieur du système GAPA, ainsi, chaque fois qu'un message arrive il est nécessaire de sélectionner la spécification idoine et de valider le message par rapport à cette dernière. Il y a ensuite un co-routinage entre le mécanisme de sélection de la spécification adaptée et de l'analyseur syntaxique. Enfin, GAPAL est basé sur une syntaxe dérivée de la *metasyntaxe* ABNF, ce qui permet une réutilisabilité simple et naturelle des spécifications que l'on peut trouver dans les normes. Dans les différentes publications, l'évaluation est marginale et ne permet pas d'affirmer la viabilité d'une telle solution. De plus, il n'existe pas de distribution du langage ni d'exemples complets de spécifications.

Mélange [Mad07] est un projet d'architecture pour construire des implémentations de protocoles qui intègre des méthodes formelles pour améliorer la robustesse des serveurs : 1) des systèmes de type statiques de la famille ML ; 2) du *model-checking* pour vérifier de manière exhaustive les propriétés de sûreté des serveurs ; et 3) de la meta-programmation générative pour exprimer des contraintes haut-niveau pour les tâches spécifiques au domaine comme l'analyse syntaxique des paquets et la définition de machines à états. Pour ces deux derniers aspects, l'architecture s'appuie sur deux langages dédiés. D'une part, le Meta Packet Language [MHD+07] (MPL), un langage dédié à la description de format des protocoles. Un générateur produit du code sans erreur de type pour appréhender le trafic réseau. D'autre part, le Statecall Policy Language pour la définition de machine à états. Une spécification MPL consiste en une succession de champs nommés et typés avec une liste optionnelle d'attributs. Le compilateur génère un analyseur syntaxique en O'caml à la fois efficace et sans erreur de types. Les serveurs SSH et DNS basés sur cette architecture présentent un meilleur débit et une latence plus faible que leurs équivalents en C. Ainsi, les bénéfices induits par le code généré par le compilateur de MPL surpassent les surcoûts induits par le ramasse-miettes automatique et la vérification dynamique des bornes.

4.5 Bilan

Dans la première section de ce chapitre, nous avons décrit différents travaux visant à prouver la viabilité de l'utilisation de langages de programmation avancés pour la programmation système et réseau. Le constat est cependant clair, la programmation système et réseau s'effectue toujours en C. La deuxième section présente une instance d'approche verticale à l'implémentation d'applications réseau. L'approche tout-intégré permet de programmer une certaine catégorie d'applications partageant un support protocolaire cohérent et commun. Toutefois, une approche verticale restreint de manière drastique la dimension de la famille de programmes considérée. Dans le reste du chapitre, nous avons présenté des approches horizontales qui peuvent être utilisés pour développer le support protocolaire d'applications réseau. Nous avons démontré qu'un langage permettant la description de données semi-structuré était trop générique pour être utilisé dans le contexte d'applications réseaux. Nous nous sommes ensuite focalisés sur des solutions réellement spécifiques mais ces solutions souffrent

de quelques défauts qui grèvent leur adoption.

Chapitre 5

Démarche suivie

Dans le chapitre 2, nous avons dressé un portrait des communications réseaux en mettant en évidence la nécessité d'un support protocolaire robuste et efficace pour les applications réseaux. Les langages dédiés, décrits dans le chapitre 3, peuvent permettre d'apporter de telles garanties comme l'a notamment prouvé l'approche Melange [MHD+07]. Nous avons recensé dans le chapitre 4 des travaux importants concernant des approches visant à systématiser le développement logiciel pour les communications réseaux.

Nous rappelons, dans ce chapitre, les problématiques auxquelles doit faire face le programmeur du support protocolaire d'applications réseaux. Ces problématiques sous-tendent toute notre démarche depuis la conception jusqu'à la réalisation d'une approche langage au développement du support protocolaire d'applications réseaux que nous présentons par la suite.

5.1 Problématique

La figure 5.1 représente le code pour le support de l'en-tête `Via` de SIP et illustre le type de code rencontré dans les applications réseaux pour le support protocolaire. Ce code, très éloigné de la spécification du protocole, forme la dernière couche logicielle traversée par un message protocolaire avant que les informations qu'il véhicule soient traités par la logique de l'application. Ainsi, il représente l'interface entre l'application et le monde extérieur.

Plusieurs constatations motivent l'étude des problèmes posés par le développement du support protocolaire d'applications réseaux :

- Le support protocolaire représente une grande part d'une application réseau tant en termes de volume de code qu'en temps de traitement. Une récente étude [CEE04] a montré que la seule analyse syntaxique de messages SIP représentent environ 25% du temps total de traitement d'un message dans le cas d'un serveur mandataire. Durant nos travaux, nous avons constaté que le volume de code nécessaire au support protocolaire représente environ un cinquième voire un quart du volume de code total. Ainsi, cette couche logicielle est une partie importante des applications réseaux.
- Nous avons présenté dans la section 2.3 la complexité du développement du support protocolaire : complexité intrinsèque des protocoles applicatifs et complexité du code à écrire pour le support de ces protocoles. La complexité du code vient principalement du fossé à combler entre la spécification formelle du protocole et le langage de programmation utilisé pour développer le support de ce protocole.
- Nous avons également mis en avant le fait que le support protocolaire d'applications réseaux

```
for (tmp=p;tmp<end;tmp++){
  switch(*tmp){
    case ' ':
      switch(state){
        case FIN_HIDDEN:
        case FIN_ALIAS:
          param->type=state;
          param->name.len=tmp-param->name.s;
          state=L_PARAM;
          goto endofparam;
        case FIN_BRAN
    CH:
        case FIN_TTL:
        case FIN_MADDR:
        case FIN_RECEIVED:
        case FIN_RPORT:
        case FIN_I:
          param->type=state;
          param->name.len=tmp-param->name.s;
          state=L_VALUE;
          goto find_value;
[...]
```

FIG. 5.1 – Partie du code permettant l'analyse de l'en tête *Via* dans SER

est une couche logicielle critique. De nombreuses attaques ciblent les applications réseaux, les serveurs notamment, en cherchant à exploiter des failles de sécurité présentes dans le support protocolaire des applications visées. En effet, le support protocolaire demeure l'ultime rempart entre une application réseau et le monde extérieur.

– Les différents travaux de recherche que nous avons présentés dans le chapitre 4 et notamment dans la section 4.4 n'apportent pas de solutions satisfaisantes pour le développement du support protocolaire d'applications réseaux pour des protocoles textuels. Le paradigme, l'architecture ou le code généré et les contraintes inhérentes au langage cible choisi (mémoire et prévisibilité notamment) par ces travaux empêchent leur adoption massive par les programmeurs. Ainsi, l'outil Melange [MHD+07] semble être le plus avancé mais son approche basé sur un langage fonctionnel est rédhibitoire pour une diffusion massive.

Ces différentes problématiques couplées à l'omniprésence d'applications réseaux justifient l'intérêt d'une nouvelle approche pour le développement du support protocolaire d'applications réseaux. Nous présentons maintenant la démarche globale que nous avons suivie pour la conception et la réalisation de cette nouvelle approche.

5.2 Démarche globale

L'approche que nous proposons pour faciliter le développement du support protocolaire d'applications réseaux est fondée sur l'utilisation d'un langage dédié, tel que présenté dans le chapitre 3. La figure 5.2 illustre le processus de développement que nous définissons. Une spécification du support protocolaire est tout d'abord écrite à partir de la grammaire formelle du protocole. Cette spécification est ensuite analysée afin de déceler d'éventuelles erreurs ou incohérences. Puis, le code nécessaire à la manipulation de messages protocolaires est automatiquement généré à partir de la spécification. Enfin, le programmeur d'une application réseau peut utiliser ce code pour gérer les échanges entre la logique de l'application et le monde extérieur.

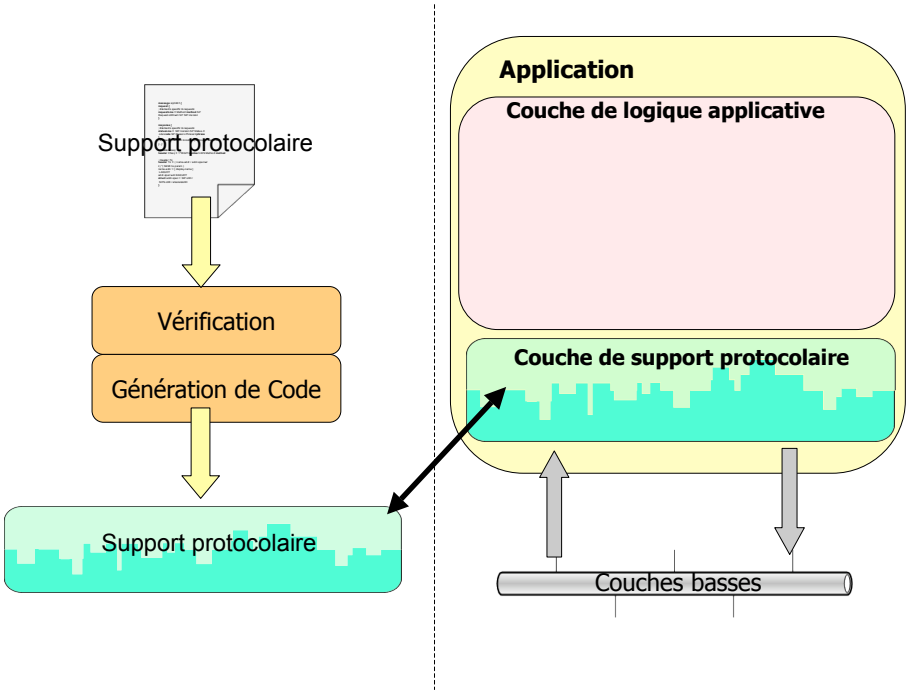

FIG. 5.2 – Processus de développement du support protocolaire

La première étape de notre démarche consiste à concevoir un langage dédié à la spécification du support protocolaire d'une application réseau. Comme nous l'avons suggéré dans le chapitre 3, nous avons mené une analyse de domaine et de famille [Par76, Nei80, Wei96] qui vise à identifier les abstractions et les notations appropriées. La seconde étape définit l'ensemble des vérifications à effectuer au niveau des spécifications écrites dans notre langage. Ces vérifications portent principalement sur la cohérence de la spécification, comme la détection de doublons et d'omissions. La troisième étape concerne la phase de génération de code. Différents schémas de compilation peuvent offrir différentes implémentations. Il est ainsi possible d'obtenir une version effectuant une analyse syntaxique très stricte ou d'intégrer certaines optimisations permettant d'améliorer les performances à l'exécution. La dernière étape de notre démarche consiste à évaluer la qualité du support protocolaire généré à l'aide de notre outil. Nous nous intéressons à deux critères d'évaluation : la robustesse et la performance à l'exécution. Nous cherchons ainsi à mesurer l'impact potentiel de notre approche sur le développement d'applications réseaux.

Dans le reste de ce document, nous détaillons les différentes étapes de la démarche que nous venons de présenter. Tout d'abord, nous décrivons le développement d'un langage dédié pour la spécification du support protocolaire d'une application réseau : depuis l'analyse de domaine jusqu'aux différentes implémentations d'un compilateur. Puis, nous présentons les différentes expérimentations que nous avons réalisées afin d'évaluer la robustesse et la performance à l'exécution d'applications réseaux reposant sur notre approche.

Deuxième partie

Approche proposée

Chapitre 6

Analyse de domaine et de famille

La phase de conception est une étape cruciale dans le processus de développement d'un langage. En effet, les résultats de cette conception constituent les fondations de l'implémentation réelle du langage. Pour mener à bien cette conception, nous avons présenté diverses méthodologies existantes dans le chapitre 3. Ces méthodologies guident tout le processus de conception en s'appuyant sur une analyse du domaine et de la famille de programmes considérés. Ce chapitre présente l'analyse de domaine et de famille que nous avons conduite pour guider la conception de notre langage. À partir de diverses sources d'informations, les points communs entre les différents membres de la famille de programmes sont extraits. Les variations entre ces mêmes membres sont analysées pour définir les bornes du langage à créer. Enfin, nous illustrons les contraintes de conception préalablement mentionnées dans le domaine que nous considérons.

6.1 Sources d'informations

La source d'informations primordiale lors du développement d'un langage dédié est la communauté d'utilisateurs visée. De nombreux échanges sur les listes de diffusion de projets libres nous ont permis de peaufiner la conception de notre langage pour intégrer les attentes des utilisateurs potentiels. Nous avons ensuite recensé une série de *bonnes* pratiques que nous avons extraites du code source de nombreuses bibliothèques et applications réseaux comme des serveurs HTTP, des serveurs mandataires SIP, HTTP et RTSP, et enfin divers clients SIP, HTTP et RTSP. Cette étude a été étayée par des informations récoltées dans la littérature [Var05]. Ces pratiques ont été introduites dans notre schéma de compilation sous la forme de patrons de code. Nous avons également référencé des attaques visant à exploiter des erreurs classiques telles le débordement de tableaux ou le déréférencement de pointeurs nuls, du support protocolaire d'applications réseaux. Nous avons utilisé ces informations lors de l'évaluation de robustesse du support protocolaire généré. Ces différents travaux ont été corroborés par l'étude approfondie des RFC des différents protocoles considérés.

6.2 Points communs

La parenté existant entre différents membres d'une même famille de programmes se caractérise par un ensemble de points communs. Ces points communs permettent la réutilisabilité recherchée lors de la conception d'un langage dédié. Les points communs peuvent être séparés en deux catégories [Nei80] : les *parties communes* et les *assemblages communs*. On parle de parties communes

lorsque les mêmes briques logicielles sont assemblées différemment. Ainsi, les bibliothèques de fonction systématisent la réutilisation de parties communes. Par opposition, on parle d'assemblages communs lorsque des briques logicielles différentes sont assemblées suivant les mêmes motifs. Le concept de canevas a été développé dans ce but.

Un langage dédié fournit dans sa chaîne de compilation à la fois les assemblages communs et les parties communes du domaine considéré. La réutilisation d'assemblages communs peut être occultée par l'adoption d'un langage déclaratif. En effet, le générateur associé intègre les assemblages communs sous forme de patrons de code qui encadrent la génération de code. En effet, la génération du code propre à une spécification particulière se fait en respectant ces assemblages. Les parties communes sont généralement exposées à l'utilisateur dans la syntaxe du langage comme par exemple par l'introduction de types primitifs au niveau du langage.

6.2.1 Grammaires

Les parties communes et les assemblages communs sont des points communs des membres de la famille de programmes. Cependant, à un niveau plus élevé, les protocoles applicatifs considérés présentent tous un point commun, à savoir leur formalisation syntaxique à l'aide de la *metasyntaxe* ABNF. Une spécification ABNF est, à l'instar d'une spécification BNF, composée d'un ensemble de règles qui forment la grammaire du protocole. La figure 6.1 reprend la grammaire complète de l'ABNF, elle-même formulée en ABNF. Ainsi, une règle (clause `rule`) est définie par un ensemble d'éléments (clause `elements`). Les éléments peuvent être concaténés ou alors alternés. Un élément est une référence à une règle par son nom, plusieurs éléments groupés par des parenthèses. Un élément peut également décrire un ensemble de valeurs sous forme binaire, ou sous forme de caractères, voire d'entiers.

À l'origine, chaque RFC définissait son propre formalisme de description de messages. La RFC 2234 [CO97] extrait de ces redéfinitions une spécification complète et indépendante qui fait depuis office de référence. Toutes les RFC depuis 1997 se fondent sur cette définition pour établir la grammaire du protocole concerné. Ainsi, l'ABNF est un formalisme qui grâce à sa simplicité s'est largement répandu pour devenir un standard reconnu.

6.2.2 Nature des messages

Nous l'avons vu, les communications réseaux s'effectuent à l'aide d'un échange de requêtes et de réponses associées. La nature d'un message, requête ou réponse, est généralement immédiatement identifiable par des différences évidentes. Ainsi dans le cas de HTTP, les requêtes diffèrent des réponses entre autres par la première ligne (ligne de commande) comme nous pouvons le voir à la figure 6.2.

La première ligne d'une requête HTTP (Figure 6.3a) contient une méthode qui indique au serveur le traitement à effectuer. Cette première ligne indique aussi sur quelle ressource ce traitement doit être effectué en précisant l'identifiant de la ressource concernée (`Request-URI`). Par opposition, la première ligne d'une réponse HTTP contient le statut du traitement qui a été demandé par la requête. Le statut du traitement peut être soit une réussite, soit un échec et la cas échéant la raison de cet échec.

```
rulelist       =  1*( rule / (*c-wsp c-nl) )

rule           =  rulename defined-as elements c-nl
                             ; continues if next line starts
                             ;  with white space
rulename       =  ALPHA *(ALPHA / DIGIT / "-")

defined-as     =  *c-wsp ("=" / "=/") *c-wsp
                            ; basic rules definition and
                            ;  incremental alternatives

elements       =  alternation *c-wsp

c-wsp          =  WSP / (c-nl WSP)

c-nl           =  comment / CRLF
                      ; comment or newline

comment        =  ";" *(WSP / VCHAR) CRLF

alternation    =  concatenation
                      *(*c-wsp "/" *c-wsp concatenation)

concatenation  =  repetition *(1*c-wsp repetition)

repetition     =  [repeat] element

repeat         =  1*DIGIT / (*DIGIT "*" *DIGIT)

element        =  rulename / group / option /
                         char-val / num-val / prose-val

group          =  "(" *c-wsp alternation *c-wsp ")"

option         =  "[" *c-wsp alternation *c-wsp "]"

char-val       =  DQUOTE *(%x20-21 / %x23-7E) DQUOTE
                             ; quoted string of SP and VCHAR
                                without DQUOTE

num-val        =  "%" (bin-val / dec-val / hex-val)

bin-val        =  "b" 1*BIT
                         [ 1*("." 1*BIT) / ("-" 1*BIT) ]
                              ; series of concatenated bit values
                              ; or single ONEOF range

dec-val        =  "d" 1*DIGIT
                         [ 1*("." 1*DIGIT) / ("-" 1*DIGIT) ]

hex-val        =  "x" 1*HEXDIG
                         [ 1*("." 1*HEXDIG) / ("-" 1*HEXDIG) ]

prose-val      =  "<" *(%x20-3D / %x3F-7E) ">"
                             ; bracketed string of SP and VCHAR
                                without angles
                             ; prose description, to be used as
                                last resort
```

FIG. 6.1 – Grammaire ABNF (formulée en ABNF)

```
Request = Request-Line
          *(( general-header
          | request-header
          | entity-header ) CRLF)
          CRLF
          [ message-body ]
```

6.2a. ABNF des requêtes HTTP

```
Response = Status-Line
           *(( general-header
           | response-header
           | entity-header ) CRLF)
           CRLF
           [ message-body ]
```

6.2b. ABNF des réponses HTTP

FIG. 6.2 – ABNF des messages HTTP

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```

(a) Première ligne d'une requête

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

(b) Première ligne d'une réponse

FIG. 6.3 – Premières lignes des messages HTTP

6.2.3 Structure des messages

Les protocoles textuels, hormis la distinction classique requête/réponse, présentent des similarités dans la structure de leurs messages. Deux grandes familles s'opposent : les protocoles *mono-lignes* et les protocoles *à la* HTTP. Les protocoles mono-lignes sont des protocoles simples et anciens, comme SMTP [Pos82], pour lesquels les messages se terminent par un retour chariot et un saut de ligne. Cette seule ligne de texte contient à la fois la nature de la commande à effectuer et l'ensemble des paramètres de cette commande. Par abus de langage, les requêtes des protocoles mono-lignes sont appelés commandes.

Des protocoles comme SIP ou Stomp [sto], sont apparus récemment dans le sillage du protocole HTTP. Ces protocoles *à la* HTTP sont généralement plus complexes que les protocoles mono-lignes dans le sens où une requête entraîne plusieurs actions au niveau du serveur concerné. Un message, que ce soit une requête ou une réponse, comme le montre la figure 6.2, est constitué d'une première ligne différenciée que nous avons présentée à la section 6.2.2, puis d'un ensemble d'en-têtes qui respectent le même format global, le nom de l'en-tête suivi du caractère deux-point puis la valeur de cet en-tête.

Chaque en-tête est un paramètre à l'échange en cours. Par exemple, le message SIP représenté à la figure 2.3 contient notamment l'identifiant de l'appelant (`From`) et de l'appelé (`To`). Néanmoins, tous les en-têtes n'ont pas la même incidence sur la communication. Ainsi, si certains sont cruciaux pour le bon fonctionnement des opérations, d'autres sont données à titre informationnel. Il est important de noter à la figure 6.2 que certains en-têtes peuvent s'avérer spécifiques aux requêtes (`r`equest-header) ou aux réponses (`r`esponse-header) quand d'autres sont génériques (`g`eneral-header et `e`ntity-header).

Les messages se terminent par une charge utile optionnelle, le *body*, qui peut remplir divers offices. Cette charge utile peut véhiculer des paramètres pour la logique de l'application, comme la négociation multimédia pour SIP. Elle peut également être utilisée pour le transport de données. Dans le cas de HTTP, les pages Internet sont rapatriées par ce biais. Le format de cette charge utile est non-contraint, le type de données véhiculées doit donc généralement être précisé dans un en-tête spécifique.

Cette section a permis d'identifier les différents points communs à considérer dans notre domaine. D'une part, les protocoles sont spécifiés en utilisant la *metasyntaxe* ABNF, la syntaxe du langage

que nous concevons doit s'en approcher pour minimiser l'effort de développement. D'autre part, les messages des différents protocoles présentent des similitudes. Ainsi, la première ligne d'un message permet d'identifier la nature du message : requête ou réponse. Le reste du message, quelque soit sa nature, est constitué d'une suite d'en-têtes qui représentent des paramètres à l'échange en cours. Certains de ces en-têtes peuvent être spécifiques aux requêtes ou aux réponses. Ces différents points communs doivent remonter au niveau de l'utilisateur du langage et doivent donc être exposés par la syntaxe du langage.

6.3 Variations

Dans une analyse de domaine et de famille, nous cherchons d'abord les points communs pour maximiser la réutilisabilité. Cependant, si une analyse de domaine s'intéresse seulement aux points communs, l'analyse de famille, elle, identifie les variations entre les différents membres d'une même famille de programmes. La raison principale pour s'intéresser aux variations entre membres d'une même famille est qu'un des objectifs des langages dédiés est de fournir des abstractions de haut niveau qui soient spécifiques au domaine considéré. Un objet qui varie en fonction des membres de la famille suggère la conception d'une abstraction permettant de le décrire. Il faut néanmoins que ces variations soient bornées pour pouvoir définir un domaine de variation.

Dans notre contexte, il existe deux sources de variation : le protocole considéré et la logique applicative qui requiert le support de ce protocole.

6.3.1 Protocole

Les protocoles que nous considérons sont spécifiés à l'aide du même formalisme et présentent des similitudes dans la structuration des messages. Néanmoins, au delà de leur syntaxe qui varie, les protocoles varient entre eux de deux manières. Tout d'abord, concernant les valeurs, les protocoles peuvent exprimer des bornes en termes de maximum pour des entiers ou de longueur pour une chaîne de caractères. Dans le cas de HTTP par exemple, les codes de réponse doivent être compris entre 100 et 699. Ensuite, les protocoles varient entre eux par des contraintes structurelles au niveau des messages. Dans le cas de SIP, certains en-têtes sont obligatoires dans certains contextes particuliers.

Cependant, la *metasyntaxe* ABNF permet seulement de définir la syntaxe du protocole, c'est-à-dire l'ensemble des valeurs que peuvent prendre les différents éléments d'un message protocolaire et l'agencement de ces éléments. Ainsi, les variations que nous venons d'introduire peuvent s'avérer difficilement exprimables en ABNF. En effet, ABNF est purement syntaxique dans le sens où il n'y a aucune interprétation sémantique des valeurs définie. Une valeur finale ou constante est exprimée entre guillemets ou à l'aide d'expressions régulières, il n'y a donc pas, par conséquent, de distinction possible entre la définition d'une valeur entière ou d'une chaîne de caractères. Sans interprétation sémantique, il devient difficile de préciser des contraintes telles que des bornes sur les entiers. Cependant, l'utilisation de telles bornes est nécessaire pour la définition des codes de réponse valides du protocole HTTP. Pour décrire de telles contraintes en ABNF, il faut soit énumérer tous les codes possibles entre 100 et 699, soit écrire une expression régulière relativement complexe. De plus, il n'existe pas en ABNF de moyen simple pour contraindre la longueur d'un élément d'un message ; il est seulement possible de définir le nombre de répétitions maximales d'un même motif. Or, généralement, le fait de ne pas contraindre ces longueurs (notamment des identifiants de ressources) exposent à des attaques de type *buffer overflow*. Ce type d'attaques consiste à accéder à une zone mémoire en dehors des limites autorisées, ce qui peut aboutir entre autres à l'exécution de code arbitraire.

ABNF ne permet pas non plus de préciser facilement des contraintes structurelles au niveau du message. Par exemple, la manière classique de décrire un ensemble de choix possibles est d'utiliser une énumération d'alternatives. Or, dans ce cas, il est impossible d'exprimer que certaines alternatives sont obligatoires et doivent apparaître au moins une fois dans le message. Par exemple, les en-têtes de messages SIP sont décrits dans la RFC par une énumération. Ainsi, la présence obligatoire de certains en-têtes n'apparaît pas clairement dans la spécification ABNF de SIP. Pour ce faire, il faudrait remanier toute la grammaire du protocole au risque de la rendre illisible. Une autre contrainte structurelle est la dépendance entre différents éléments du message. En SIP, l'en-tête `CSeq` doit reprendre la méthode qui est précisée à la première ligne pour que le message soit valide. Cette contrainte est précisée en langage naturel dans la spécification car il est impossible de préciser une telle contrainte en ABNF.

6.3.2 Application réseau

Une source de variations entre membres de la famille considérée est la nature de la logique applicative. Nous l'avons vu dans le chapitre 2, il existe différents types d'applications réseaux qui ont chacun une logique applicative adaptée. Notamment, dans la section 2.3.2, nous nous sommes efforcés de mettre en parallèle la logique de l'application et l'accès au message courant. En effet, suivant l'application considérée, la *vue* désirée du message est différente. La vue est la vision que l'application a du message, c'est-à-dire la structure de données par laquelle elle a accès à ce message. Une vue est plus ou moins complète et plus ou moins structurée. Une vue est complète quand elle restitue l'entièreté du message par opposition à une vue partielle. La structure d'une vue dépend du niveau de décomposition du message souhaité. En effet, un message peut être vu comme à tiroirs, chaque élément pouvant être décomposé en sous-éléments jusqu'à obtenir les éléments ou valeurs finaux.

Dans les implémentations étudiées, la logique de l'application manipule la même structure de données que le support protocolaire. Or, le support protocolaire ne peut s'affranchir d'analyser tout le message. Généralement, les structures de données que l'on trouve dans les applications sont un compromis entre sûreté et flexibilité. Certaines parties du message bénéficient d'une vue très structurée car cela a été requis par le support protocolaire tandis que d'autres parties, jugées moins critiques, sont représentés par une longue chaîne de caractères. Néanmoins, l'hyper-structuration de la vue d'un message peut être très préjudiciable pour le programmeur. En effet, l'approche classique est de représenter un message sous la forme d'enregistrements imbriqués, une hyper-structuration entraîne donc une sur-imbrication de structures. Ainsi, pour atteindre un élément particulier du message, le programmeur doit, soit utiliser une notation pointée assez longue, soit faire appel successivement à plusieurs fonctions et stocker les éléments intermédiaires successifs. La programmation devient alors fastidieuse résultant en un code verbeux qui nécessite de nombreuses copies mémoire.

Nous avons vu dans cette section les différentes variations que le langage que nous concevons doit être en mesure d'exprimer, les variations de valeur, les contraintes structurelles et la vue du message. Néanmoins, il est important de garder à l'esprit les contraintes qui sous-tendent la conception de tout langage dédié : une programmation facilitée, une réutilisation systématique, une sûreté améliorée et l'optimisabilité.

6.4 Contraintes de conception

Dans le chapitre 3, à la section 3.2.2, nous avons présenté les différents bénéfices escomptés au développement d'un nouveau langage dédié. L'analyse de domaine et de famille que nous avons décrite dans ce chapitre permet d'obtenir les paramètres d'entrée à la conception de ce nouveau langage

dédié. À la lumière de cette analyse, nous illustrons la prise en compte des contraintes de conception d'un langage dédié dans notre domaine.

6.4.1 Programmation plus facile

Pour faciliter la programmation, il est nécessaire de fournir à l'utilisateur un ensemble approprié de notations et d'abstractions. Un point important résultant de notre analyse est le fossé qui existe entre la *metasyntaxe* ABNF et le code à écrire pour respecter une grammaire. Combler ce fossé manuellement est un exercice très difficile, ce qui favorise l'apparition de nombreuses erreurs. Idéalement, une spécification haut niveau de la grammaire devrait permettre de générer le support protocolaire. Néanmoins, la seule spécification ABNF d'un protocole ne permet pas de définir tout le support protocolaire. Il faut ajouter à cette spécification, un ensemble d'informations pour exprimer des contraintes structurelles sur les messages. En effet, notre analyse nous a permis d'identifier certaines contraintes spécifiées de manière informelle en langage naturel dans le texte des normes de protocoles.

6.4.2 Réutilisation systématique

La réutilisation dans un langage dédié peut prendre plusieurs formes, réutilisation de code et réutilisation de schémas. La réutilisation de code est rendue automatique par l'environnement d'exécution. Dans le cas d'un langage déclaratif, la réutilisation de schémas est cachée à l'utilisateur grâce au fossé qui existe entre le langage dédié et le langage cible. Il existe une autre forme de réutilisation plus évidente : la réutilisation de conception. La réutilisation de conception consiste à représenter la connaissance du domaine en définissant un ensemble d'opérations spécifiques au domaine. La spécification, ou le programme, écrite dans le langage dédié doit être organisé de façon à indiquer clairement la décomposition d'un problème ainsi que les décisions de conception à prendre à chaque niveau de cette décomposition. Dans notre étude, nous avons identifié la notion de message. Un message est soit une requête, soit une réponse dont on distingue la nature par sa première ligne.

6.4.3 Sûreté améliorée

Un des intérêts à développer un langage dédié est de rendre certaines propriétés décidables. En effet, un langage dédié rend possible certaines analyses qui seraient indécidables ou irréalisables dans un contexte général par des restrictions et/ou en enrichissant les informations fournies par l'utilisateur. Néanmoins, dans le cas d'un langage déclaratif, la sûreté est garantie davantage par construction que par vérification puisqu'un langage déclaratif n'a pas de logique opérationnelle.

La principale source d'erreurs dans le développement du support protocolaire d'applications réseaux provient du fossé entre le formalisme ABNF et le langage C que le programmeur doit combler à la main. En montant le niveau d'abstraction du langage au niveau des utilisateurs, nous éliminons, d'une part, toutes les erreurs inhérentes à C comme une mauvaise gestion de la mémoire et d'autre part nous garantissons la conformité du support protocolaire par rapport à la grammaire spécifiée. Les vérifications qui sont ensuite effectuées garantissent la correction de la grammaire et la prédictabilité de sa compilation. Ainsi, nous vérifions des règles de cohérence telles que l'obligation de définir toute règle référencée exactement une seule fois.

6.4.4 Optimisabilité

Le support protocolaire d'une application réseau doit se montrer le plus efficace possible en temps et en mémoire pour laisser le maximum de ressources à l'exécution de la logique applicative. Nous

devons donc prendre en compte les performances lors de notre processus de conception. Lors de notre étude des applications réseaux reposant sur des protocoles applicatifs textuels existants, nous avons remarqué certaines bonnes pratiques peu répandues car relativement difficiles à mettre en œuvre systématiquement. Par exemple, SER implémente une analyse syntaxique étagée (ou paresseuse) qui permet de ne pas valider le message dans son ensemble mais seulement les éléments du message requis. Cette stratégie conduit à nettement améliorer le temps de traitement et la consommation des ressources mémoire. L'utilisation d'un langage dédié doit systématiser de telles pratiques.

6.5 Bilan

Ce chapitre décrit l'analyse de domaine et de famille de programmes que nous avons menée pour la conception d'un langage dédié au développement du support protocolaire d'applications réseaux. Cette analyse nous a permis de définir un cahier des charges pour le langage à concevoir.

Nous avons mesuré l'importance du formalisme utilisée pour la spécification de protocoles. La *metasyntaxe* ABNF est un élément central dans le développement de notre approche. L'adoption de ce formalisme facilite la tâche de programmation en évitant au programmeur de manipuler un langage bas niveau.

Nous avons également identifié différentes notions transverses aux différents protocoles considérés comme la notion de message. Dans les protocoles que nous considérons, les messages respectent une structure commune. De plus, un message ne peut être que de deux natures, soit requête, soit réponse. Ces différents points communs sont repris comme invariants dans notre langage.

Une fois les différents points communs identifiés, nous nous sommes attachés à définir un ensemble de variations entre les différents supports protocolaires. Les variations que nous avons identifiées proviennent de deux sources : le protocole considéré et la logique applicative associée. Le protocole considéré, hormis la grammaire spécifiée, dicte un ensemble de contraintes structurelles qui lui sont inhérentes. La logique de l'application, quant à elle, doit pouvoir définir la vue du message qu'elle souhaite manipuler. Cette vue du message doit pouvoir être définie indépendamment des structures de données internes utilisées par l'analyseur syntaxique du support protocolaire pour valider un message.

Chapitre 7

Zebu

Une analyse de domaine fournit un ensemble de directives nécessaires au développement d'un langage dédié. L'analyse que nous avons menée et présentée dans le chapitre précédent repose sur deux sources : les protocoles et les applications considérés. Les résultats de cette analyse ont dirigé la conception d'un langage dédié appelé Zebu pour le développement du support protocolaire d'applications réseaux.

D'une part, le langage doit être en mesure d'exprimer la structure des messages protocolaires et les contraintes inhérentes aux protocoles textuels. D'autre part, le langage doit permettre de configurer le code généré pour l'adapter aux besoins de l'application réseau considérée.

Nous l'avons vu, la couche de support protocolaire d'une application réseau est à la frontière de deux mondes : le monde protocolaire et le monde applicatif. Ainsi, certaines caractéristiques de cette couche sont dictées par le protocole considéré, par opposition à certaines caractéristiques que l'on souhaite voir dépendre de l'application. Par conséquent, la présentation de notre langage est divisée en deux parties. La première partie décrit les aspects protocolaires d'une spécification Zebu. La seconde partie s'attache à présenter les aspects applicatifs introduits dans le langage, permettant de diriger la génération de la couche de support protocolaire pour satisfaire les attentes du programmeur de l'application réseau.

7.1 Introduction

Dans le chapitre précédent, nous avons noté l'importance du formalisme ABNF utilisé pour la spécification de la syntaxe des messages protocolaire. Ce formalisme allie simplicité et puissance d'expression : simplicité inhérente à un formalisme réduit à seulement quelques constructions et puissance d'expression permise par le niveau d'abstraction fourni. La simplicité d'une spécification contraste, d'ailleurs, à la fois avec la difficulté du processus de traduction d'une telle spécification vers un langage procédural comme C ainsi qu'avec la complexité du code résultant. Notre approche se fonde sur ABNF pour capitaliser sur l'existant, la spécification en ABNF de la grammaire des messages d'un protocole étant toujours disponible dans la norme de ce protocole. L'objectif est de minimiser l'intervention du programmeur par rapport à la spécification ABNF initiale pour éviter les erreurs. Le programmeur doit seulement restructurer la spécification pour séparer explicitement les éléments spécifiques selon la nature du message et annoter les différents éléments du message qu'il souhaite voir être accessibles par la couche de logique applicative. Ces annotations vont notamment permettre au programmeur de définir la vue du message qu'il souhaite manipuler au niveau de la couche applicative de l'application réseau.

7.2 Aspects protocolaires d'une spécification Zebu

Les aspects protocolaires d'une spécification Zebu proviennent de trois sources. La première, la plus générale, découle du fait que nous considérons seulement des protocoles applicatifs textuels avec une certaine structure de message. Les deux suivantes sont spécifiques à un protocole particulier. D'une part, la spécification ABNF du protocole définit les aspects syntaxiques du protocole. D'autre part, le texte associé à cette spécification, généralement dans une RFC, définit un ensemble de contraintes concernant la structure globale de messages.

7.2.1 Socle commun

L'étude de différents protocoles applicatifs nous a permis d'identifier trois concepts clés : message, requête et réponse. Un message est une unité de transfert au niveau de la couche application. Si un message transite d'un client à un serveur, on parle de requête ; dans l'autre sens, on parle de réponse. La figure 7.1 montre un extrait de la grammaire du protocole HTTP formalisant la distinction entre requêtes et réponses. Ainsi, un message est formellement défini comme étant soit une requête soit une réponse.

```
HTTP-message    = Request | Response      ; HTTP/1.1 messages
```

FIG. 7.1 – Nature d'un message HTTP

Un message HTTP comme nous l'avons vu à la section 6.2.3, que ce soit une requête ou une réponse respecte la même structure globale, première ligne, différents en-têtes puis charge utile optionnelle, dont la spécification ABNF est représentée à la figure 7.2. Cette structure de message est le dénominateur commun entre les différents protocoles *à la* HTTP.

```
generic-message = start-line
                          *(message-header CRLF)
                          CRLF
                          [ message-body ]
start-line      = Request-Line | Status-Line
```

FIG. 7.2 – Structure globale d'un message pour les protocoles *à la* HTTP

Ces différents points communs forment les fondations du langage. Ainsi, une spécification Zebu explicite cette structure comme le montre la figure 7.3. Le bloc `message` englobe toute la spécification, il permet de délimiter la spécification. Nous l'avons précédemment évoqué, certains éléments du message ne peuvent apparaître que dans des requêtes ou que dans des réponses. La conséquence de cet état de fait au niveau du langage est l'apparition de deux blocs spécifiques, le bloc `request` et le bloc `response`. Le bloc `request` (lignes 3-5) permet de définir des éléments de message censés n'apparaître que dans des requêtes. Le bloc `response` (lignes 6-8) permet de définir des éléments de message spécifiques aux réponses. Cette vue en blocs permet d'éviter la spécification de règles intermédiaires pour isoler certains en-têtes comme, par exemple, les règles `request-header` et `response-header` auxquelles font référence les figures 6.2a et 6.2b respectivement. De plus, par ce biais, la séparation entre éléments spécifiques à la requête et éléments spécifiques à la réponse devient claire.

```
1  message sip3261 {
2   request {     ; Request only
3    requestLine = Method SP  Request-URI  SP SIP-Version
4  [...]
5   }
6   response {    ;Response only
7    statusLine = SIP-Version SP Status-Code SP Reason-Phrase
8  }
9   Method = INVITEm / ACKm / OPTIONSm / BYEm
10                / CANCELm / REGISTERm / extension-method
11  extension-method = token
12  INVITEm = %x49.4E.56.49.54.45 ; INVITE in caps
13  [...]
14  Request-URI = SIP-URI / SIPS-URI / absoluteURI
15  header CSeq = 1*DIGIT LWS Method
16  [...]
17  header To { "to" / "t" } =
18     ( name-addr / addr-spec ) *( SEMI to-param )
19  }
```

FIG. 7.3 – Structure d'une spécification Zebu

7.2.2 Aspects spécifiques au protocole considéré

Le socle commun que nous venons de présenter a été utilisé pour définir la structure globale d'une spécification écrite dans le langage Zebu. Nous présentons maintenant les aspects spécifiques à un protocole particulier dont va découler le *corps* de la spécification.

7.2.2.1 Éléments extraits de la spécification ABNF

Pour passer d'une spécification ABNF à une spécification Zebu, le programmeur doit séparer les éléments spécifiques aux requêtes de ceux spécifiques aux réponses mais aussi effectuer quelques transformations au niveau de certaines règles. Nous présentons ce processus, après un court rappel sur le formalisme ABNF, en nous appuyant sur l'exemple du protocole SIP.

La figure 7.4 représente un extrait de la spécification ABNF du protocole SIP. Une spécification ABNF consiste en un ensemble de règles de dérivation, chacune de ces règles définissant un ensemble d'alternatives, chaque alternative étant séparée par le symbole '`/`'. Une alternative est une séquence de terminaux et de non-terminaux. On parle de non-terminal quand une règle fait référence à une autre règle de dérivation. Les terminaux, quant à eux, peuvent être de trois types, chaînes de caractères constantes insensibles à la casse, chaînes de caractères constantes sensibles à la casse et expressions régulières. Un terminal peut également être un ensemble d'alternatives de terminaux. D'un point de vue syntaxique, une chaîne de caractères entre guillemets est insensible à la casse. Les chaînes de caractères sensibles à la casse doivent être spécifiées comme une séquence explicite des codes ASCII des caractères de cette chaîne, comme dans la règle `INVITEm` à la figure 7.4, ligne 3. Une expression régulière représente à la fois la structure du terminal et les caractères autorisés dans la valeur de ce terminal.

ABNF inclut une forme générale de répétition, $n{\star}m$`X`, qui indique qu'au moins n et au plus m occurrences du terminal ou non-terminal `X` doivent apparaître. ABNF définit aussi des abréviations comme $n\star$ pour $n{\star}\infty$, $\star n$ pour $0{\star}n$, $\star$ pour $0{\star}\infty$, n pour $n{\star}n$. Par conséquent, $1{\star}$`DIGIT` dans la règle `CSeq` (ligne 7) représente une séquence de chiffres de longueur au moins 1. Les crochets sont utilisés comme abréviation pour $0{\star}1$ et permettent ainsi d'exprimer si certains éléments sont

```
1  Request-Line     = Method SP Request-URI SP SIP-Version CRLF
2  Method           = INVITEm / ACKm / OPTIONSm / BYEm
                        / CANCELm / REGISTERm / extension-method
3  INVITEm          = %x49.4E.56.49.54.45 ; INVITE in caps
4  Request-URI      = SIP-URI / SIPS-URI / absoluteURI
5  SIP-Version      =  "SIP" "/" 1*DIGIT "." 1*DIGIT
6  extension-method = token
7  CSeq             = "CSeq" HCOLON 1*DIGIT LWS Method
8  LWS              =  [*WSP CRLF] 1*WSP ; linear whitespace
9  SWS              =  [LWS] ; sep whitespace
10 HCOLON           =  *( SP / HTAB ) ":" SWS
```

FIG. 7.4 – Extrait de l'ABNF définissant la syntaxe d'un message SIP (RFC 3261)

optionnels.

Le processus de traduction est assez simple et systématique. Le programmeur Zebu annote la règle définissant la syntaxe de la première ligne d'une requête avec le mot-clé `requestLine`, la règle définissant la syntaxe de la première ligne d'une réponse avec le mot-clé `statusLine` et les règles définissant la syntaxe de chaque en-tête avec le mot-clé `header`. La réorganisation de la spécification de la grammaire ABNF en différents blocs chacun spécifique à une nature de message permet de s'abstenir de la définition de non-terminaux pour les premières lignes et les en-têtes. En effet, la présence de non-terminaux pour ces différents éléments provenait de la nécessité d'exprimer la structure globale d'un message. Dans le cas de Zebu, cette structure globale du message est connue, ces règles intermédiaires ne sont donc plus nécessaires dans l'état actuel. L'effort de transformation entre la spécification ABNF de départ et la spécification Zebu se situe à ce niveau, dans le passage des non-terminaux pour les lignes de commande et en-têtes vers des règles Zebu. Ainsi, par exemple, la règle ABNF pour le non-terminal `Request-Line` représentée à la figure 7.4 ligne 1 est transformée dans la règle Zebu suivante :

```
requestLine = Method SP Request-URI SP SIP-Version
```

Dans le cas d'un en-tête de message, la description de la clé (le *nom* de l'en-tête) est déplacée de la partie droite de la règle vers la partie gauche, où elle remplace l'identifiant de la règle, résultant dans une règle dont la structure suggère une paire clé/valeur. Par exemple, la règle ABNF `CSeq` à la ligne 7 de la figure 7.4 est réorganisée dans la règle Zebu suivante (figure 7.3 ligne 15) :

```
header CSeq = 1*DIGIT LWS Method
```

Le délimiteur `HCOLON` est également abandonné, car il représente une constante du protocole. Ce délimiteur demeure un paramètre de Zebu pour, par exemple, permettre la prise en charge de protocoles utilisant le symbole '=' comme séparateur entre le nom d'un en-tête et sa valeur. Certains en-têtes, tel que l'en-tête SIP `To`, peuvent être représentés par plusieurs clés, la clé étant à choisir parmi un ensemble. Ce procédé est généralement utilisé pour définir des formes abrégées des clés. Dans ce cas, l'en-tête est nommé ; le nom de l'en-tête est suivi pour une spécification ABNF des différentes variantes, entre accolades, tel qu'illustré à la ligne 17 de la figure 7.3. De manière semblable à ABNF, la définition des clés, tout comme n'importe quelle autre chaîne de caractères spécifiée par une grammaire Zebu, est insensible à la casse.

7.2.2.2 Contraintes exprimées de manière informelle

Le texte accompagnant la spécification ABNF d'un protocole, notamment dans les RFC, fournit de manière informelle des informations complémentaires à la seule syntaxe du protocole que l'on peut trouver dans la spécification ABNF. Typiquement, le texte d'une RFC indique dans quelles conditions un certain en-tête peut apparaître ou s'il est obligatoire. Il peut également spécifier quels en-têtes sont en lecture seule et lesquels sont modifiables. On parle d'en-têtes en lecture seule quand tous les messages tout le long d'une communication partagent un ensemble d'éléments en commun. Enfin, certaines contraintes sur les valeurs des en-têtes peuvent également être exprimées. Le programmeur Zebu doit annoter les règles ABNF correspondantes avec ces contraintes. Les contraintes sont spécifiées entre accolades à la fin d'une règle de grammaire. Les contraintes atomiques possibles sont qu'un en-tête peut être obligatoire (`mandatory`) ou qu'un en-tête peut apparaître plus d'une fois (`multiple`). Par exemple, dans la spécification du protocole SIP, l'en-tête `To` est spécifié comme étant obligatoire (Figure 7.5, ligne 21). Des contraintes plus complexes peuvent être exprimées en utilisant des expressions booléennes *à la* C. Par exemple, le texte de la RFC de SIP spécifie que quand le message est une requête, la méthode mentionnée au niveau de la première ligne du message doit être la même que celle spécifiée dans l'en-tête CSeq. Cette contrainte est décrite à la figure 7.5, ligne 4.

Certaines contraintes sur les en-têtes sont spécifiques soient aux requêtes, soient aux réponses. Par conséquent, le programmeur Zebu peut naturellement grouper de telles contraintes dans le bloc approprié. Dans le cas du protocole SIP, le bloc `request` (lignes 2 à 6) indique par exemple que l'en-tête `Max-Forwards` est obligatoire. La contrainte sur la correspondance entre les méthodes de la première ligne d'une requête et l'en-tête CSeq est également spécifiée dans ce bloc. Pour plus de clarté, les contraintes du bloc `response` (ligne 7 à 9) ont été éludées.

7.3 Aspects applicatifs d'une spécification Zebu

La figure 7.3 a permis d'appréhender la structure globale d'une spécification Zebu et de se familiariser avec les différences entre ABNF et Zebu. ABNF présente quelques lacunes que Zebu comble par la possibilité d'exprimer des contraintes sur la structure globale d'un message. Ainsi, une spécification englobe tous les aspects protocolaires nécessaires à la génération d'un support protocolaire robuste. Néanmoins, le programmeur souhaite pouvoir injecter dans ce support protocolaire des aspects spécifiques à une couche applicative particulière. Le programmeur en annotant les éléments de la grammaire ABNF peut décrire la vue du message qu'il souhaite retrouver au niveau de la couche applicative. Cette vue est une structure de données, définie par les différentes annotations, qui regroupe les éléments du messages auxquels la couche applicative souhaite avoir accès en lecture ou écriture. La définition de la structure de données dicte la génération de code, des structures de données aux fonctions talons en passant par l'analyseur syntaxique.

Les annotations fournies par Zebu permettent de définir différentes facettes d'une même spécification. Nous distinguons quatre facettes que nous distinguerons par la suite : vue du message, fonctions utilitaires de lecture, fonctions utilitaires d'écriture et analyseur syntaxique.

La figure 7.5 représente la spécification Zebu équivalente à l'extrait de la spécification ABNF du protocole SIP de la figure 7.4 présentant les différentes annotations que nous allons présenter.

7.3.1 Vue du message

L'analyse de domaine que nous avons menée a permis de définir des structures de données génériques pour tout message d'un protocole applicatif textuel, tout en-tête et toute ligne de commande.

```
1  message sip3261 {
2   request {; Request only
3    requestLine = Method:method SP Request-URI:uri SP SIP-Version
4    header CSeq {CSeq.method == requestLine.method}
5    header Max-Forwards {mandatory}
6   }
7   response {; Response only
8    statusLine = SIP-Version SP Status-Code:code SP Reason-Phrase:rphrase
9  }
10  enum Method = INVITEm / ACKm / OPTIONSm / BYEm / CANCELm / REGISTERm
11                      / extension-method
12  extension-method = token
13  INVITEm = %x49.4E.56.49.54.45 ; INVITE in caps
14  struct Request-URI = SIP-URI / SIPS-URI / absoluteURI {lazy}
15  uint16 Status-Code = Informational / Redirection / Success / Client-Error
16                       / Server-Error / Global-Failure / extension-code
17  uint16 Global-Failure = "600"/ "603"/ "604"/ "606"
18  uint16 extension-code = 3DIGIT {extension-code>=100 && extension-code<=699}
19  header CSeq = 1*DIGIT:number as uint32 LWS Method:method
20  header Max-Forwards = 1*DIGIT:value as uint32 {mandatory}
21  header To {"to" / "t"} = (name-addr / addr-spec:uri) *( SEMI to-param )
22                           {mandatory, ReadOnly}
23  name-addr = [display-name] LAQUOT addr-spec:uri RAQUOT
24  struct addr-spec = SIP-URI / SIPS-URI / absoluteURI {lazy}
25  [...]
```

FIG. 7.5 – Extrait d'une spécification Zebu pour le protocole SIP

Ces structures sont ensuite complétées par la vue particulière souhaitée pour une application. Cette vue est en quelque sorte un filtre appliqué au message textuel qui transite sur le réseau. Le programmeur annote les éléments du message auxquels il souhaite avoir accès. De cette manière, l'application n'a accès qu'aux éléments du message souhaités. De plus, l'accès à ces éléments est facilité par la différenciation du message brut de sa vue structurée. Dans les approches classiques, les structures de données reflètent exactement le message brut, chaque élément étant représenté comme un champs d'une structure englobante. Ainsi, généralement, une structure est créée par non-terminal. Or, pour atteindre un élément il faut traverser de nombreuses structures. Par conséquent, l'accès à un élément particulier se fait par l'intermédiaire d'une notation pointée arbitrairement profonde.

Les annotations en partie droite permettent de sélectionner les éléments du message qui seront disponibles. Le programmeur doit seulement faire suivre un élément (que ce soit un non-terminal ou non) par un symbole deux-points et un identifiant pour le rendre disponible au niveau de la structure de données associée. Cette annotation engendre la création par le compilateur Zebu d'un champ reprenant le même identifiant dans la structure de données, qui représente l'élément annoté. Par exemple, à la ligne 3 de la figure 7.5, le programmeur Zebu a indiqué que l'application requiert l'utilisation de la méthode et de l'URI de la ligne de commande d'une requête. Ainsi, dans la structure de données représentant la ligne de commande d'une requête deux champs seront présents : le champs `method` et le champs `uri` tout deux sous la forme d'une chaîne de caractère. À la fin de l'analyse syntaxique d'un message, ces champs représente les éléments du message correspondants. Par défaut, si aucun élément d'une ligne de commande ou d'un en-tête n'est annoté, la structure de données associée consiste seulement en un nouveau type structuré qui contient comme champ unique un pointeur vers la position de départ dans le message de la ligne de commande ou de l'en-tête considéré.

Les annotations en partie gauche définissent les types sous lesquels les éléments sont regroupés et par ce biais à quel niveau ces éléments seront disponibles. En partie gauche, le programmeur an-

note les différentes règles de la spécification qu'il souhaite retrouver comme des types dans la vue du message. Les différents éléments annotés en partie droite sont *aspirés* et remontés dans la vue du message jusqu'à rencontrer une telle annotation. Il peut annoter une règle avec `struct`, comme la règle `addr-spec` à la ligne 24 de la figure 7.5. Pour chaque occurrence de la règle `addr-spec` dans le message reçu, l'analyseur syntaxique instancie une structure de données dédiée. Cette structure de données dédiée regroupe tous les éléments du message annotés obtenus par la dérivation de la règle `addr-spec`. Une règle consistant en un ensemble d'alternatives de terminaux peut être annotée avec `enum`. Cette annotation a pour conséquence de créer un nouveau type énuméré regroupant l'ensemble des types possibles pour chaque alternative. Ainsi, la règle `Method`, ligne 10, est une énumération des différentes méthodes possibles. Le fait d'annoter cette règle comme une énumération va permettre de caractériser le message selon sa méthode. Enfin, l'annotation `union` est précisée dans le cas d'alternatives de non-terminaux. À l'instar des types unions en C, cette annotation permet de représenter, sous un seul type de données, divers types hétérogènes. Par contre, les unions Zebu permettent de savoir quel constructeur de type a été utilisé. Ainsi, le programmeur de l'application réseau peut s'assurer que les champs de l'union sont accédés correctement. Ces différentes annotations en partie gauche, tout comme les mots-clés `header`, `requestLine` et `statusLine` agissent comme des points d'arrêt à la remontée des éléments dans la vue du message lors de l'aspiration et conduisent à la création de nouveaux types. Une structuration plus fine est ainsi obtenue par l'introduction de types intermédiaires.

Les éléments extraits d'un message sont par défaut représentés sous la forme d'une structure de données contenant deux champs : la position de départ de l'élément considéré et la longueur de cet élément. Souvent, cependant, l'application va nécessiter l'utilisation de cette valeur sous une autre forme, comme un entier par exemple. Dans les couches de support protocolaire étudiées, cette conversion est souvent mal gérée car la valeur à convertir n'est pas validée par rapport aux valeurs acceptées par la fonction de conversion. Cette non-vérification conduit à la levée d'une erreur qui n'est généralement pas traitée et peut donc compromettre la couche de logique de l'application. Le programmeur Zebu peut préciser lorsqu'il souhaite qu'un élément annoté du message soit converti en un autre type que chaîne de caractères par rapport sous forme d'un entier. Pour ce faire, il doit seulement faire suivre l'identifiant de l'élément annoté avec la construction `as` suivi du type souhaité. Par exemple, à la ligne 19, le nombre spécifié par l'en-tête CSeq est spécifié comme étant un `uint32`. La règle de définition d'un terminal peut également être précédée de la spécification d'un type comme aux lignes 15, 17 et 18. Cette contrainte de type permet non seulement de représenter cet élément sous la forme d'un entier dans les structures de données générées mais également de contraindre la valeur de cet élément par des assertions vérifiées à l'exécution. Ainsi, le champs `number` de l'en-tête CSeq sera strictement inférieur à 2^{32}.

7.3.2 Analyseur syntaxique et fonctions utilitaires de lecture

La couche applicative n'utilise pas directement les structures de données déclarées dans la spécification Zebu et présentes dans le support protocolaire. En effet, l'application y accède par des fonctions (on parle de fonctions talons pour *stubs*) générées par le compilateur telles que celles représentées à la figure 7.6. Ces fonctions sont de différents types. On y distingue une fonction, `sip3261_init`, permettant d'initialiser une structure de données représentant un message et d'autres permettant de commander directement l'analyseur syntaxique telles que `sip3261_parse` qui en est le point d'entrée et `sip3261_parse_headers` déclenchant l'analyse syntaxique des en-têtes d'un message. Des fonctions talons sont également générées pour obtenir les informations résultantes de l'analyse syntaxique telles que `sip3261_isRequest` et `sip3261_isResponse` pour déterminer la nature

d'un message ou `sip3261_get_RequestLine` qui retourne une structure de données représentant la première ligne d'une requête et reprenant les informations annotées.

```
sip3261 sip3261_init();
void    sip3261_parse(sip3261, char *, int);
void    sip3261_parse_headers(sip3261, E_Headers);
void    sip3261_parse_addr_spec(T_Lazy_addr_spec);
T_bool  sip3261_isRequest(sip3261);
T_bool  sip3261_isResponse(sip3261);
T_Str   sip3261_Option_Str_getVal(T_Option_Str);
T_RequestLine sip3261_get_RequestLine(sip3261);
T_RequestLine  sip3261_get_RequestLine(sip3261);
T_header_From  sip3261_get_header_From(sip3261);
T_Method       sip3261_RequestLine_getMethod(T_RequestLine);
T_MethodEnum   sip3261_Method_getType(T_Method);
T_Str          sip3261_Method_getValue(T_Method);
T_Lazy_addr_spec sip3261_header_From_getUri(T_header_From);
T_addr_spec    sip3261_Lazy_Addr_spec_getParsed(T_Lazy_addr_spec);
T_Option_Str   sip3261_Addr_spec_gethost(T_addr_spec);
```

FIG. 7.6 – Fonctions talons générées par le compilateur Zebu

Le compilateur Zebu peut être configuré pour générer un analyseur syntaxique plus ou moins strict. En faisant varier ce curseur, on obtient une analyse syntaxique plus ou moins fine du message. Par exemple, le programmeur peut vouloir accepter des messages potentiellement mal formés à condition que les éléments qu'il a annotés soient corrects. De cette manière, seules les parties du message nécessaires à la bonne exécution des fonctions talons sont analysées strictement.

Les fonctions d'analyse syntaxique générées par le compilateur Zebu mettent en oeuvre une stratégie d'analyse syntaxique à deux niveaux. Un analyseur syntaxique générique gros grain ou de niveau supérieur parcourt chaque ligne du message jusqu'à atteindre la ligne de commande ou l'en-tête désiré. À ce moment, il laisse la main à un analyseur syntaxique dédié à la ligne de commande ou à l'en-tête repéré. Une fois la tâche de cet analyseur dédié terminée, l'analyseur syntaxique gros grain reprend la main pour éventuellement continuer l'analyse syntaxique dans la continuité de ce qui vient d'être fait. Il existe donc un analyseur syntaxique gros grain et une collection d'analyseurs syntaxiques dédiés, chacun de ces analyseurs syntaxiques dédiés traite un type particulier de ligne de commande ou d'en-tête. Les analyseurs syntaxiques dédiés doivent respecter à la fois la spécification ABNF ainsi que les contraintes spécifiées de manière informelle dans le texte de la RFC du protocole considéré, comme le fait que certains en-têtes sont obligatoires ou des contraintes de dépendance intra-message. Les analyseurs syntaxiques stockent les éléments du message dans les structures de données générées tout en vérifiant les contraintes exprimées au niveau de la spécification. Une fois la ligne de commande ou l'en-tête du message analysé syntaxiquement et vérifié, les éléments annotés sont convertis dans les types spécifiés et stockés dans les structures de données associées. Les valeurs des éléments nommés peuvent être ensuite accéder par l'entremise d'une fonction talon préfixée par "*get*".

Pour des éléments du message particulièrement complexes et/ou des éléments dont on ne souhaite l'analyse syntaxique que dans certaines conditions, l'analyse syntaxique peut être rendue incrémentale (ou *paresseuse*). L'annotation Zebu `lazy` spécifie qu'un élément particulier du message ne doit être analysé syntaxiquement que sur demande de l'application. Dans la spécification SIP, `Request-URI` présente cette annotation (ligne 14) tout comme la règle `addr-spec`. Une analyse syntaxique incrémentale permet d'éviter l'analyse syntaxique d'éléments complexes d'un message (dans ces cas, une URI) qui peuvent s'avérer inutiles. Par ce biais, la profondeur de la vue du message est ajustée au besoin de l'application. Concrètement, l'analyseur syntaxique dédié à la ligne de commande ou l'en-

tête reprenant cette règle note seulement la position de départ et la longueur de l'élément qui doit être analysé de manière incrémentale dans une structure de données particulière. Ensuite, l'appel à une fonction talon telle que `sip3261_Lazy_Addr_spec_getParsed` exécute l'analyse syntaxique de cet élément.

7.3.3 Fonctions utilitaires d'écriture

Hormis les fonctions de lecture qui pilotent l'analyseur syntaxique, le compilateur de Zebu génère des fonctions de modification du message. Ces fonctions de modification sont générées pour chaque élément annoté à l'instar des fonctions de lecture. Ces fonctions de modification sont utiles dans le contexte de l'implémentation d'un serveur mandataire qui doit transmettre un message après en avoir modifié certaines parties. Cependant, certains éléments d'un message ne peuvent être modifiés selon la norme. Le mot-clé `ReadOnly` permet de spécifier qu'un élément est accessible en lecture seule. Au niveau de la génération, les fonctions de modification pour cet élément ne sont pas générées.

7.4 Vérification des spécifications

Le langage Zebu que nous venons de présenter est fondé sur la *metasyntaxe* ABNF. Nous avons étendu ce formalisme en définissant une structure globale à une spécification tout en lui adjoignant diverses annotations. Ces annotations permettent, d'une part, de formaliser des contraintes spécifiées de manière informelle en marge des spécifications ABNF et, d'autre part, de laisser la possibilité au programmeur d'une application réseau de spécifier des structures de données sur mesure pour manipuler la vue du message qu'il souhaite.

Les propriétés de cohérence d'une spécification Zebu s'expriment à trois niveaux : la spécification ABNF, les contraintes de type et les contraintes structurelles.

7.4.1 Cohérence de la spécification ABNF

L'écriture d'une spécification ABNF peut s'avérer relativement complexe, et ce d'autant plus qu'il n'existe pas d'outils, notamment d'éditeurs, permettant de vérifier la cohérence d'une grammaire ABNF. Une spécification ABNF pour être cohérente doit notamment respecter les trois propriétés critiques listées ci dessous.

Absence de redéfinition. Une règle de la grammaire ne peut être définie qu'une seule fois. Il est possible d'étendre une règle à l'aide de l'opérateur '`/=`', mais il est interdit de redéfinir une règle à l'aide de l'opérateur '`=`'.

Absence d'omission. Une règle référencée doit être définie. Cependant, ABNF ne contraint pas d'ordre dans les définitions ; ainsi, il n'est pas nécessaire d'ordonner les définitions des règles selon un ordre particulier.

Absence de cycle. Une spécification ABNF ne doit pas présenter de cycles. Un cycle apparaît lorsqu'une règle peut se dériver en elle-même par une suite de dérivations successives. La présence de cycles dans une spécification posent certains problèmes de sécurité au niveau des analyseurs syntaxiques, notamment, en termes d'allocation et libération de mémoire.

7.4.2 Cohérence des contraintes

Une spécification Zebu étend une spécification ABNF par l'ajout d'annotations permettant d'exprimer des contraintes structurelles et du typage. En effet, l'annotation par le programmeur des éléments du message qu'il souhaite voir apparaître dans la vue du message peut être assimilée à du typage.

Contraintes structurelles. Les contraintes structurelles exprimées entre accolades doivent être cohérentes. Ainsi, il ne peut y avoir égalité qu'entre des éléments annotés équivalents. De plus, des contraintes spécifiques aux requêtes ne peuvent faire référence à des éléments spécifiques aux réponses, et inversement.

Typage. La spécification de la vue du message doit être correctement typée par rapport à la grammaire du message. Plus précisément, l'annotation de type doit correspondre aux valeurs potentielles d'un élément du message. Par exemple, un non-terminal ne peut être annoté comme étant un entier que s'il peut se dériver en un nombre. De même, une union ne peut exister qu'entre différentes alternatives.

7.5 Bilan

Nous avons présenté dans ce chapitre notre langage dédié Zebu, pour spécifier la couche de support protocolaire d'une application réseau. Sa conception découle de l'analyse de domaine et de famille que nous avons présentée dans le chapitre précédent. Zebu reprend la syntaxe du formalisme ABNF dédié à la spécification de protocoles. Nous avons permis aux programmeurs Zebu d'influer, par le biais d'annotations, sur le processus de génération pour adapter le support protocolaire à une application donnée et également spécifier des contraintes non-exprimables en ABNF. Nous avons également défini un ensemble de propriétés de cohérence des spécifications que vérifie un compilateur Zebu avant la génération.

Chapitre 8

Développement d'applications réseaux avec Zebu

Dans le chapitre précédent, nous avons présenté un langage dédié déclaratif appelé Zebu, permettant de générer le support protocolaire d'applications réseaux. Dans ce chapitre, nous présentons un processus de développement d'applications réseaux fondées sur l'utilisation de ce langage.

Dans un premier temps, nous définissons le domaine de l'étude en précisant le langage et les plates-formes ciblées. Ensuite, nous nous appuyons sur le chapitre 5 pour détailler le processus de développement général à suivre. Enfin, nous présentons l'étape de compilation d'une spécification Zebu. Nous présentons deux chaînes de compilation dont une ciblant plus particulièrement les systèmes embarqués.

8.1 Domaine de l'étude

L'étude que nous menons dans ce chapitre est restreinte à un seul langage, C. Le langage C, nous avons largement insisté sur ce point dans les chapitres précédents, est le langage de choix dans la programmation système et réseaux. Dans notre étude, nous avons développé un premier compilateur Zebu, appelé ZebuYacc que nous avons expérimenté sur des plates-formes Linux. Une grande partie des applications réseaux fonctionnent sur cette plate-forme et la communauté Linux fournit de nombreuses sources d'information. Nous avons ensuite fait évoluer ce compilateur et nous lui avons adjoint d'autres modules pour obtenir une chaîne de compilation complète ciblant les systèmes embarqués.

Les différents éléments présentés dans ce chapitre ne sont cependant pas spécifiques au langage C et peuvent être facilement portés sur d'autres langages de programmation. Ainsi, il semble raisonnable de cibler des langages de programmation tels que C++, Java ou encore O'Caml.

8.2 Processus de développement

La figure 8.1 illustre le processus de développement d'une application réseau fondée sur Zebu. Dans un premier temps, une spécification du support protocolaire est écrite en Zebu (partie de gauche de la figure 8.1). Une phase de vérifications est ensuite effectuée pour déceler d'éventuelles incohérences. Cette phase de vérifications terminée, le compilateur de Zebu génère alors, de façon automatique, les structures de données telles que définies par les annotations de la spécification Zebu. Le compilateur met également à disposition de l'utilisateur un ensemble de fonctions pour lire et écrire

dans les structures de données générées. Enfin, ZebuYacc génère un analyseur syntaxique respectant la spécification Zebu. Cet analyseur syntaxique s'appuie sur une infrastructure générique que nous avons mise au point et qui reprend l'ensemble des points communs que nous avons identifiés au chapitre 6. Cette infrastructure est un patron de code dans lequel ZebuYacc vient générer les fonctions d'analyse spécifiques au protocole considéré. Le programmeur d'une application réseau (écrite en C) utilise les fonctions fournies pour mettre en œuvre la manipulation de messages protocolaires.

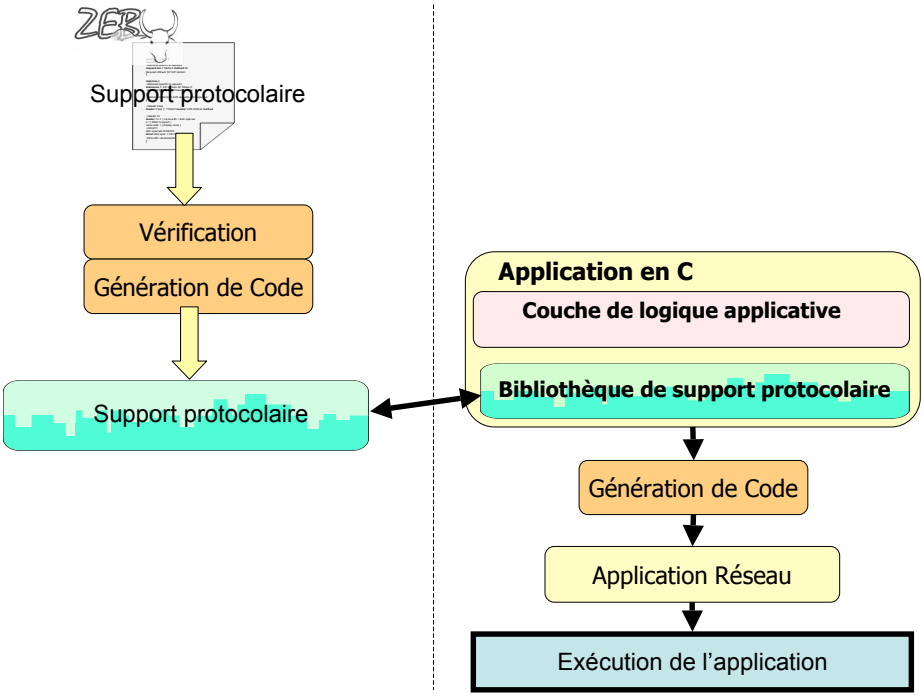

FIG. 8.1 – Processus de développement d'applications réseaux avec Zebu

8.3 Chaîne de compilation

Le compilateur Zebu vérifie la cohérence de la spécification ABNF et des annotations ajoutées par le programmeur. Ensuite, il génère les fonctions permettant à une application d'accéder aux structures de données représentant les messages préalablement analysés. Le compilateur représente environ 7250 lignes de code O'Caml. L'environnement d'exécution générique qui fournit de surcroît un ensemble de fonctions utilitaires comptent environ 1200 lignes de code C.

8.3.1 Vérifications

Bien que les RFC soient largement publiées et représentent les standards *de facto* de nombreux protocoles, nous avons trouvé quelques erreurs dans les spécifications ABNF de certaines RFC. Ce sont des erreurs simples, telles que des erreurs typographiques, mais qui contribuent à rendre le processus de traduction d'une spécification ABNF vers du code encore plus complexe. Pour détecter ces

erreurs, le compilateur Zebu vérifie certaines propriétés de cohérence basales : pas d'omission, pas de double définitions, pas de cycles.

De plus, les annotations, doivent être cohérentes par rapport à la spécification ABNF. Par exemple, à la ligne 17 de la figure 7.5, le non-terminal `Global-Failure` est annoté avec `uint16`. Ce non-terminal est spécifié comme étant un choix entre divers chaînes de caractères alternatives. Ainsi, le compilateur Zebu vérifie que chaque élément de ce choix représente un entier non signé inférieur à 2^{16}.

8.4 Développer une application avec Zebu

Le développeur définit la logique de l'application comme un programme C *ordinaire*, utilisant les fonctions talons utilitaires pour accéder aux informations relatives au contenu du message. La figure 8.2 illustre l'implémentation d'une application fondée sur Zebu qui extrait, si le message traité est une requête de type `INVITE`, l'hôte émetteur de cette requête. Ce type d'application est utile, par exemple, dans les systèmes de détection d'intrusions, qui cherchent certains motifs d'information dans les messages circulant sur le réseau considéré ou plus simplement à des fins statistiques.

```
1 sip3261 msg = sip3261_init();
2 sip3261 msg = sip3261_parse(msg, buf, len);
3 if (sip3261_isRequest(msg)) {   // Process only request messages
4   T_RequestLine requestLine = sip3261_get_RequestLine(msg);
5   if (sip3261_Method_getType(sip3261_RequestLine_getMethod(requestLine))
6         == E_INVITEm) {// Filter INVITE methods
7     sip3261_parse_headers(msg, E_HEADER_FROM); // We parse only the header From
8    T_Lazy_addr_spec l_addr_spec =
9      sip3261_header_From_getUri(sip3261_get_header_From(msg));
10    sip3261_parse_addr_spec(l_addr_spec);
11    T_Option_Str host = sip3261_Lazy_Addr_spec_getParsed(l_addr_spec);
12    if (sip3261_Option_Str_isDefined(host)) {
13         // host may be undefined in some cases, check it and log its value
14       mylog(sip3261_Option_Str_getVal(host));
15 }}}
```

FIG. 8.2 – Fragment d'une application de statistiques des messages SIP fondées sur Zebu

L'application utilise les fonctions talons générées pour accéder aux informations requises. Ces fonctions générées respectent la structure de données définie par les annotations fournies. Initialement, l'application appelle les fonctions `sip3261_Method_getType` et `sip3261_RequestLine-_getMethod` pour déterminer si le message est une requête et le cas échéant si la méthode spécifiée sur la ligne de commande est `INVITE` (lignes 5-6). L'annotation de la règle `Method` par `enum` à la ligne 10 de la figure 7.5 a engendré la génération d'une énumération reprenant toutes les alternatives comme `E_INVITEm` représentant une méthode `INVITE`. Si la requête est de type `INVITE`, l'application utilise la fonction `sip3261_parse_headers` pour effectuer l'analyse syntaxique de l'en-tête `From` (ligne 7). En effet, par défaut, l'analyse syntaxique des en-têtes est incrémentale, dans le sens où le message n'est pas analysé complètement au départ mais ligne par ligne suivant les éléments requis par l'application. Les fonctions `sip3261_get_header_From` et `sip3261_header_From_getUri` sont ensuite appelées pour extraire l'URI (ligne 9) de l'en-tête.

La ligne 24 de la spécification Zebu de SIP indique que l'analyse syntaxique de l'URI doit être *paresseuse* ou incrémentale. Par conséquent, la fonction `sip3261_Lazy_Addr_spec_g-`

`etParsed` est appelée pour demander l'analyse syntaxique de ce sous-champ (ligne 11). Après vérification que la partie hôte de l'URI est définie (ligne 12), sa valeur est extraite en utilisant la fonction `sip3261_Option_Str_getVal` à la ligne 14.

Les fonctions talons générées reflètent la structure de données sous-jacente définie par les annotations. Chaque fonction talon de lecture ou d'écriture accède à un fragment particulier d'un message.

8.5 Évolution vers une chaîne de compilation complète optimisée

Le langage Zebu permet de spécifier de manière naturelle la couche de support protocolaire. Dans cette section, nous présentons une chaîne de compilation complète dans laquelle nous avons intégré des techniques de réduction de consommation mémoire, principalement pour la cible des systèmes embarqués. À l'intérieur de cette chaîne de compilation, μ-ZebuYacc est une évolution de ZebuYacc prenant en compte des problématiques de consommation mémoire chères aux systèmes embarqué. μ-ZebuGen génère le code dédié à la création de message, et μ-ZebuLink permet le développement rapide d'application pour divers systèmes d'exploitation cibles.

Aujourd'hui, les protocoles applicatifs textuels sont peu représentés dans le domaine des systèmes embarqués principalement à cause de leur lourdeur supposée. Cependant, l'utilisation de tels protocoles représente des avantages notamment en termes d'inter-opérabilité comme nous le montrons dans un premier temps. Dans un second temps, nous présentons la chaîne de compilation permettant de démocratiser l'utilisation de ces protocoles dans les systèmes embarqués communicants.

8.5.1 Contexte

Il existe un fort besoin de communication réseau entre entités hétérogènes pour de nombreuses applications embarquées. Par exemple, dans le cas de la gestion technique de bâtiment (*building automation*), un ensemble de capteurs et d'actuateurs distribué à travers un bâtiment doivent communiquer avec un contrôleur central, qui décide comment réagir à diverses situations. Ainsi, si un objet précieux est retiré de son emplacement autorisé, ce contrôleur peut décider de déclencher une alarme et de verrouiller les portes. Ce type de communication requiert un protocole pour gérer cette interaction.

Historiquement, à cause des limites intrinsèques de leurs ressources, les systèmes embarqués ont fondé leurs communications sur des protocoles binaires, non-standards et spécifiques à une application [UK94]. L'utilisation de protocoles non-standards, cependant, ségrègue un système embarqué du monde extérieur et empêche la capitalisation sur des services existants ou l'utilisation d'outils existants. De plus, concevoir un protocole dans son ensemble n'est pas un processus aisé.

Ainsi, pour ouvrir les systèmes embarqués au monde extérieur, l'attention se tourne vers l'utilisation de protocoles textuels standards *à la* HTTP. Par exemple, le protocole SIP est utilisé maintenant dans des réseaux de capteurs [Kri06] et des réseaux mobiles *ad hoc* ou MANETS pour *mobile ad hoc networks* [LMHR05, SBRA07]. L'utilisation de tels protocoles permet aux systèmes embarqués d'interagir avec des services réseaux conventionnels et permet un développement rapide des applications en utilisant des outils de mise au point existants, tels que Wireshark [1] pour suivre le flux de paquets réseaux. De plus, une caractéristique de ces protocoles est leur extensibilité. Concevoir une extension permet d'adapter le protocole aux besoins des applications en ajoutant de nouvelles méthodes, de nouveaux paramètres ou du support pour de nouveaux types de charges utiles. Ainsi, de nombreuses extensions spécifiques à des applications particulières ont été développées pour le protocole SIP [AS03, CRH01].

[1] http ://www.wireshark.org

Malgré les nombreux avantages que présentent les protocoles textuels standards, supporter de tels protocoles dans un système embarqué est un réel défi. En effet, ces protocoles sont sous leur apparente simplicité assez complexes et cette complexité devient problématique dans le contexte des systèmes embarqués avec des ressources limitées. Les analyseurs syntaxiques de messages de protocoles réseaux que nous connaissons sont très gourmands en mémoire [CEE04]. En premier lieu, le code pour un analyseur syntaxique complet est volumineux à cause de la complexité des protocoles considérés. Ensuite, les analyseurs syntaxiques rencontrés abusent de copies mémoires pour traduire le message textuel transitant sur le réseau en une structure de données le représentant. Le même constat est possible concernant la création et la modification de messages. En effet, de nombreuses structures de données intermédiaires sont créées pour contenir les valeurs à modifier.

Une autre difficulté concernant l'adoption de protocoles textuels standards dans des systèmes est la difficulté d'intégration de nouvelles applications dans un système d'exploitation embarqué. Ces systèmes d'exploitation sont minimaux (seulement trois fichiers par exemple pour FreeRTOS [fre] un mini-noyau temps-réel), nécessitant que l'application soit intégrée d'une manière dédiée au système considéré. Cette intégration requiert une expertise que ne possède pas forcément les programmeurs d'applications réseaux.

Dans la suite de cette section, nous présentons la chaîne de compilation μ-Zebu, une implémentation de Zebu qui répond aux besoins des systèmes embarqués. Nous présentons les nouveaux outils de compilation que nous avons développés pour réduire les besoins mémoires du code de traitement des messages et gérer la création et modification de message. Ces outils intègrent des évolutions du premier compilateur en termes de génération de code. Un module réellement dédié au déploiement d'applications est également présenté.

8.5.2 La chaîne de compilation μ-Zebu

Les tests préliminaires effectués sur Zebu et ZebuYacc ont prouvé que Zebu combinait la facilité de construction d'un analyseur syntaxique couplée à un haut degré de robustesse. Néanmoins, le code généré consomme trop de mémoire par rapport à la quantité présente sur les systèmes embarqués typiques. Nous avons donc développé une chaîne de compilation ciblant plus particulièrement les systèmes embarqués.

8.5.2.1 μ-ZebuYacc

Le traitement de message se divise en deux étapes. Premièrement, la *validation du message* vérifie que le format du message suit la spécification de la grammaire du protocole. Ensuite, le *rendu du message* extrait des fragments sélectionnés du message et copie ces fragments dans des structures de données dédiées à l'application selon les annotations fournies. Nous présentons les optimisations que nous avons implémentées dans μ-ZebuYacc pour chacune de ces étapes.

Validation du message Le code généré par le compilateur originel, ZebuYacc, est strictement conforme à la spécification du protocole. L'analyseur syntaxique généré accepte seulement les messages strictement valides et rejettent tous les messages invalides. Cette stratégie garantit la robustesse de l'application, mais est préjudiciable concernant la taille du code [BRLM07]. Par exemple, 50% du code généré par ZebuYacc à partir de la spécification SIP est pour la validation. Pour l'architecture arm9, cela revient à 86 ko de code compilé. Cependant, si le système embarqué est utilisé dans un environnement clos et sécurisé dans lequel il peut communiquer seulement avec des entités de confiance, alors, ce niveau de robustesse peut s'avérer superflu. Ainsi, avec μ-ZebuYacc, le programmeur peut

```
[!%'*-+\x2d-.0-9A-Z_-z~]+(?:(?:[\x9]\xd\xa)*)?[\x9
 ]+)*|(?:(?:(?:[\x9]\xd\xa)*)?[\x9]+)?"(?:(?:[!#-\
x5b\x5d-~]|(?:(?:[\x9]\xd\xa)*)?[\x9]+)|\x5c[\0-\x
9\xb-\xc\xe-\x7f])*")?(?:(?:(?:[\x9]\xd\xa)*)?[\x9
]+)?<(?:sip:(?:(?:(?:[!$&-9;=?A-Z_a-z~]|%[0-9A-F][
0-9A-F])+|(?:\x2b(?:[0-9]|[(-)\x2d-.]?)*[0-9](?:[0
-9]|[(-)\x2d-.]?)*(?:;(?:[!$&-+\x2d-:A-\x5b\x5d_a-
z~]|%[0-9A-F][0-9A-F])+(?:=(?:[!$&-+\x2d-:A-\x5b\x
5d_a-z~]|%[0-9A-F][0-9A-F])+)?|;ext=(?:[0-9]|[(-)\
x2d-.]?)+|;isub=(?:[!$&-;=?-Z_a-z~]|%[0-9A-F][0-9A
-F])+)*|(?:[#*0-9A-F]|[(-)\x2d-.]?)*[#*0-9A-F](?:[
#*0-9A-F]|[(-)\x2d-.]?)*(?:;phone-context=(?:(?:(?
:[0-9A-Za-z]|[0-9A-Za-z][\x2d0-9A-Za-z]*[0-9A-Za-z
])\x2e)*(?:[A-Za-z]|[A-Za-z][\x2d0-9A-Za-z]*[0-9A-
Za-z])\x2e?|\x2b(?:[0-9]|[(-)\x2d-.]?)*[0-9](?:[0-
```

8.3a. Validation complète (fragment)

```
([^\x20]+)\x20
```

8.3b. Résultat de l'algorithme de réduction

FIG. 8.3 – Expressions régulières générées

choisir une validation totale du message, comme avec ZebuYacc, ou une *validation partielle*, auquel cas la validation est seulement effectuée pour les éléments du message annotés. En outre, le programmeur peut annoter un élément comme `nocheck` si l'application utilise cet élément mais sans nécessiter sa validation, ou pour effectuer sa validation au niveau de la couche de logique applicative.

Dans le cas où la validation de certains éléments n'est pas requise, il peut s'avérer nécessaire d'analyser partiellement le texte, pour trouver le début de l'élément annoté suivant. Pour ce faire, nous avons développé un algorithme permettant de réduire le nombre d'états dans la partie de l'automate d'analyse syntaxique dédiée aux éléments du message non-validés. Cet algorithme identifie, tout d'abord, des jetons qui doivent précéder un élément annoté tout en n'apparaissant pas à l'intérieur de cet élément. Ces jetons sont considérés comme des *bornes* qui doivent être atteintes avant l'élément désiré. Les états représentant l'analyse syntaxique des éléments du message autres que ces bornes sont fusionnés, ce qui peut résulter dans une réduction très importante de la taille de l'automate.

En guise d'exemple concret, dans l'éventualité où l'on ait annoté le sous-champ `Request-URI` avec `nocheck` à la figure 7.5, l'analyseur syntaxique doit trouver le début et la fin de la valeur de l'URI. Comme le sous-champ `Request-URI` est suivi par un espace et qu'un espace ne peut apparaître dans une URI, différents états de l'automate pour l'analyse syntaxique de l'URI sont fusionnés pour collecter seulement les caractères différents d'espace. La figure 8.3a montre un fragment de l'expression régulière générée par μ-ZebuYacc à partir de l'automate qui est créé quand la validation complète est activée pour le sous-champ `Request-URI`. L'expression régulière complète fait 12784 caractères de longueur. L'expression régulière générée à partir de l'automate optimisé par la fusion de différents états est représentée à la figure 8.3b et illustre parfaitement le facteur de réduction que cette optimisation permet.

Rendu du message Une spécification Zebu contient des annotations indiquant les éléments du message devant être rendus disponibles à l'application ainsi que leurs structures de données associées. Les fonctions talons générées par ZebuYacc copient ces valeurs dans des structures de données locales, où elles peuvent être accédées par l'application. Le plus fréquemment, cependant, l'application utilise la valeur d'un sous-champ telle quelle, sous sa forme de chaînes de caractères. Dans ce cas, nous pouvons éviter les coûts en temps et espace mémoire de la copie de la chaîne de caractères dans une structure de données intermédiaire. En effet, les protocoles que nous considérons sont encodés textuellement. Ainsi, les messages sont déjà sous la forme de chaînes de caractères. Par conséquent,

la seule différence entre la représentation *verbatim* d'un sous-champ dans le message originel et la représentation de ce sous-champ attendue par l'application est un caractère `'\0'` à la fin. Si un sous-champ n'est pas suivi immédiatement (sans séparateur) par un autre sous-champ demandé *verbatim*, alors la fonction talon générée par μ-ZebuYacc peut simplement insérer un `'\0'` à la suite de ce sous-champ directement dans le message originel, en recouvrant le caractère précédemment présent. Cette fonction enregistre ensuite l'adresse du point de départ du sous-champ dans la structure du message dans une structure de données locale, plutôt que d'enregistrer l'adresse d'un tampon frais contenant une copie de la valeur du sous-champ désiré.

Cette stratégie, cependant, corrompt la structure du message originel, qui peut être requise pour transmettre ou créer un nouveau message. Pour remédier à ce problème de reconstruction du message originel, la fonction talon générée stocke également le caractère qui a été recouvert dans sa structure de données locale, technique connue sous le nom d'échange de caractères (*character swapping*), notamment implémentée dans Lex [LS75]. En utilisant cette technique, aucune allocation mémoire ni copie n'est nécessaire.

8.5.2.2 μ-ZebuGen

Traditionnellement, une bibliothèque de support protocolaire fournit à l'utilisateur un ensemble de fonctions pour créer, initialiser et modifier des parties d'un message [exo03]. Lors d'une modification d'un message, les données à ajouter sont de taille arbitraire, taille qui ne peut être connu statiquement. Par conséquent, les fonctions fournies opèrent sur une représentation en mémoire qui est multi-niveaux, par l'imbrication de structures permettant ainsi un accès direct à chacun des sous-champ. Une fois l'ensemble des modifications effectué, l'utilisateur commande la traduction de cette structure de données vers un message textuel. Ce message est stocké dans un tampon accédé par les couches sous-jacentes pour l'envoi du message. Cependant, cette stratégie requiert de nombreuses copies mémoires et de nombreuses structures intermédiaires. Ainsi, elle n'est pas adaptée pour une utilisation dans les systèmes embarqués.

Par opposition, μ-ZebuGen ne crée pas une structure de données multi-niveaux pour représenter le message, mais simplement collecte des informations relatives aux modifications du message. Concrètement, μ-ZebuGen fournit des fonctions pour créer un nouveau message ou copier un message existant, et des fonctions pour mettre à jour les différents éléments d'un message. Ces fonctions de mise à jour ne modifient pas réellement le message, mais stockent les modifications à effectuer sous la forme d'un *lump*. Un *lump* est une structure de données décrivant le type de modification à effectuer, la modification elle-même, où cette modification doit s'appliquer dans le tampon du message, et l'effet de cette modification sur la taille du message. Les *lumps* sont stockés dans une liste classée selon la position à l'intérieur du message à laquelle la modification doit s'appliquer. μ-ZebuGen fournit divers types de *lumps* : pour ajouter, supprimer ou remplacer un sous-champ d'un message, et pour ajouter, supprimer ou remplacer un en-tête complet. Les *lumps* permettent de minimiser la création et l'allocation de structures de données intermédiaires pour représenter les différents éléments du message à modifier.

Les *lumps* sont seulement appliqués à un message lorsque ce message est préparé à être envoyé. À ce moment, un nouveau tampon est alloué ; sa taille correspond à la taille du message originel en tenant compte des effets sur la taille du message de chaque *lump* accumulé. Le début du message originel jusqu'à la position du premier *lump* à appliquer est copié dans ce tampon, en prenant soin de restaurer les caractères échangés. La modification décrite par le *lump* est alors effectuée. Ensuite, le processus itère sur la liste des *lumps*, à partir de la fin du fragment du message qui a été modifié par l'application du *lump* précédent, jusqu'à complétion de ce nouveau message. Enfin, l'application

peut sauvegarder ce tampon et subséquemment le mettre à jour, si le même message ou de proches variantes doivent être envoyés plusieurs fois.

8.5.2.3 μ-ZebuLink

Le code généré par μ-ZebuYacc et μ-ZebuGen s'interface avec un support d'exécution sous-jacent regroupant des aspects bas-niveau tels qu'utiliser des *sockets* réseaux, prendre en charge le *multi-threading*, gérer les allocations mémoire, *etc.*. Le support disponible pour ces opérations peut varier d'un système d'exploitation à un autre, ainsi, intégrer une application fondée sur μ-Zebu dans certains systèmes d'exploitation peut s'avérer être une tâche laborieuse. De plus, certains systèmes d'exploitation pour les systèmes embarqués, comme FreeRTOS, nécessitent que l'application soit directement intégrée dans l'implémentation du système. Cette intégration demande un haut degré d'expertise et réduit la portabilité de l'application.

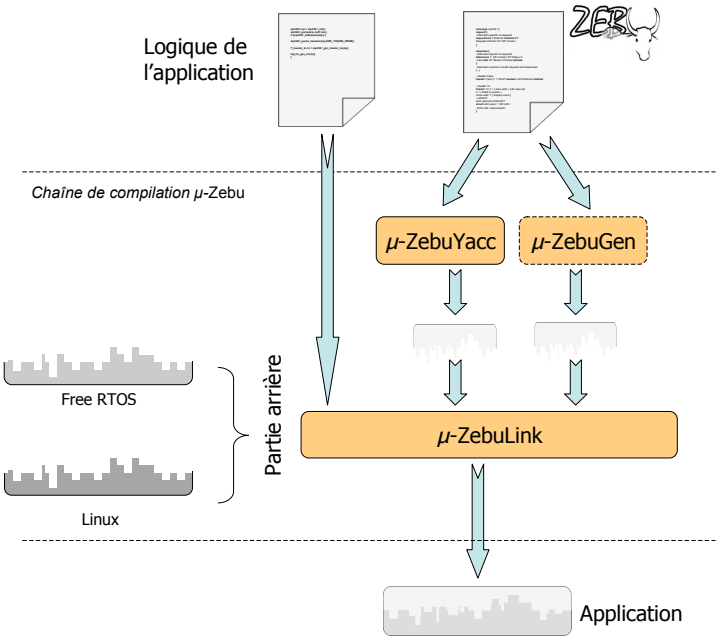

FIG. 8.4 – L'utilisation de μ-ZebuLink à l'intérieur de la chaîne de compilation μ-Zebu

Pour surmonter ces difficultés, nous avons développé un lieur, μ-ZebuLink, tel qu'illustré dans la figure 8.5 qui prend en entrée le code de l'application et le code généré par μ-ZebuYacc et μ-ZebuGen et génère une application complète dédiée à un système d'exploitation particulier. Pour le moment, nous avons implémenté deux arrière-plans pour μ-ZebuLink, un qui génère une application qui peut fonctionner sur Linux et un qui génère une version personnalisée de FreeRTOS qui intègre l'application. Ces arrière-plans incluent, tous deux, des composants génériques et du code pour personnaliser ces composants génériques selon la spécification d'un protocole donné.

Un exemple de composant générique qui est inclus dans les deux arrière-plans est un serveur de *sockets* minimal qui supporte TCP ou UDP. Le but de ce serveur de *sockets* est de convertir un

message provenant du réseau dans une chaîne de caractères qui peut être prise en charge par le code de traitement du message généré par μ-ZebuYacc. Dans le cas d'un protocole de transport orienté session, tel que TCP, le marqueur de fin du message est la fin de la session au niveau transport. Cependant, une telle information n'est pas disponible dans un protocole orienté datagramme tel qu'UDP. Par conséquent, le serveur de *sockets* doit traiter les messages entrants pour déterminer quand la fin du message a été atteinte. Dans ce cas, la personnalisation selon la spécification d'un protocole donné est requise pour indiquer comment déterminer la longueur du message. Par exemple, le protocole SIP s'appuie sur l'en-tête `Content-Length` pour spécifier la longueur de la charge utile du message. μ-ZebuLink personnalise donc un serveur de *socket* générique avec le code généré par μ-ZebuYacc pour détecter cet en-tête particulier et en déduire la taille du message.

8.6 Bilan

Nous avons décrit dans ce chapitre le processus de développement d'une application réseau fondé sur l'approche Zebu. Ce chapitre a permis d'appréhender concrètement le langage Zebu et de se familiariser avec les outils de compilation existants. Nous avons présenté le compilateur originel ZebuYacc puis son évolution, μ-ZebuYacc. μ-ZebuYacc vient s'insérer dans une chaîne de compilation complète, μ-Zebu, qui intègre diverses optimisations pour réduire l'espace mémoire et le temps consommé lors du traitement d'un message.

Troisième partie

Évaluation

Chapitre 9

Robustesse

Dans les deux derniers chapitres, nous avons détaillé l'approche que nous proposons. Nous avons présenté le langage Zebu avec ses différentes propriétés et comment développer une application réseau fondée sur ce langage, en utilisant le code généré par un compilateur. Dans ce chapitre, nous évaluons l'impact de l'utilisation de Zebu sur la robustesse du support protocolaire d'applications réseaux [BRLM07]. Nous présentons dans un premier temps la technique d'analyse que nous avons utilisée pour évaluer la robustesse des analyseurs syntaxiques fournis avec divers supports protocolaires. Cette technique, inspirée de la technique appelée *analyse de mutations* permet de calculer la couverture des différentes implémentations par rapport à une spécification en envoyant des messages erronés ou non qui peuvent s'avérer problématiques. Nous définissons un ensemble de règles de mutations des messages en nous inspirant, notamment d'une RFC [SHJ+06] recensant des types de messages problématiques. Nous décrivons ensuite une série d'expérimentations que nous avons menées sur différents supports protocolaires d'applications réseaux.

9.1 Technique utilisée

L'analyse de mutations [DLS78] fut introduite à la fin des années soixante-dix pour tester des programmes. Le principe général de cette approche est de dériver un programme sain en un mutant par injection d'une faute en modifiant une construction du programme sain de départ. Ces modifications ne sont pas appliquées au hasard mais respecte un ensemble de *règles de mutation*. Ces règles sont généralement établies à partir des erreurs fréquemment commises par les programmeurs [ADH+89].

Plus précisément, l'analyse de mutations se fonde sur l'existence d'une *classe* de fautes. Elle fournit un ensemble d'opérateurs, appelés opérateurs de mutation, qui modélisent une ou plusieurs fautes présentes dans la classe de fautes identifiée. Un opérateur de mutation est appliqué au programme que l'on souhaite tester. Ainsi, ce programme est transformé dans un programme similaire, bien que différent, connu comme un *mutant*. En général, l'application d'un opérateur de mutation peut générer un ou plusieurs mutants. Si le programme de départ contient plusieurs entités qui se trouvent dans le domaine de l'opérateur de mutation, alors l'opérateur est appliqué à chacune de ces entités. Chacune de ces applications de l'opérateur de mutation génère un mutant distinct. Par exemple, considérons un opérateur de mutation qui supprime une déclaration du programme de départ. Toutes les déclarations dans ce programme se situent dans le domaine de cet opérateur. Ainsi, à l'application de cet opérateur, autant de mutants que de déclarations seront générés. Chaque mutant présentera toutes les déclarations du programme initial excepté celle (unique) supprimée par l'opérateur de mutation. Une fois généré et compilé, un mutant est exécuté en utilisant un certain jeu de tests. Si pour un seul test de

ce jeu de tests, le résultat du mutant diffère du résultat produit par le programme initial, on dit que le mutant est *tué* (détecté). La proportion des mutants qui *meurent* (qui sont détectés) pendant l'analyse de mutations indique à quel point le programme initial est couvert par le jeu de tests.

Dans notre étude nous voulons mesurer la couverture des assertions vérifiées lors de la phase de validation des messages reçus. Une application réseau robuste doit accepter tous les messages valides par rapport à la spécification, pour fournir un service continu, et rejeter (ou au moins détecter) tous les messages invalides, pour éviter la corruption de son état interne. L'analyseur syntaxique est l'interface entre l'application et le monde extérieur, il représente la *ligne de front* dans le traitement des messages réseaux. Par conséquent, notre technique diffère de l'analyse de mutations par l'application des opérateurs de mutation. Ces opérateurs de mutation ne sont pas appliqués au programme mais aux messages en entrée pour valider le comportement de l'analyseur syntaxique. Un message sain est modifié selon diverses règles pour vérifier le comportement induit par l'introduction d'une erreur.

9.2 Règles de mutations

Les conséquences d'un support protocolaire défaillant dans sa détection de messages invalides ou dans son analyse de certains messages valides mais non-usuels peuvent s'avérer dramatiques. La plate-forme matérielle hébergeant l'application réseau peut même être compromise. Généralement, les erreurs engendrées par une mauvaise analyse syntaxique vont de l'arrêt inopiné de l'application à la prise de contrôle de la plate-forme par un attaquant malintentionné. Divers rapports de correction ont également montré que des erreurs dans le support protocolaire d'applications réseaux peuvent subsister longtemps sans être détectées. Ainsi, le serveur de diffusion multimédia Helix [hel] a présenté sur près de quatre versions une erreur dans l'analyse syntaxique des URIs permettant d'exécuter du code arbitraire sur le serveur[1]. Le but de notre étude est donc de confronter le support protocolaire d'une application réseau à un ensemble de messages délibérément pernicieux pour valider le comportement induit.

Nous avons fondé notre analyse sur un ensemble de règles de mutation déduites de la RFC 4475 [SHJ+06] qui définit un ensemble de messages de test visant à valider les implémentations de la version actuelle du protocole SIP. Nous nous sommes focalisés sur les messages ciblant les analyseurs syntaxiques. Ce document n'a pas la prétention de cataloguer toutes les catégories de messages SIP invalides, ni d'explorer tous les messages SIP valides mais non-usuels. Son objectif est de se focaliser sur des cas qui ont causé des problèmes d'inter-opérabilité ou qui présentent des caractéristiques particulièrement défavorables si elles sont gérées d'une manière inappropriée. Ce document est donc une graine de plan de test et pas un plan de test en lui-même.

Cette RFC définit une dizaine de types de messages valides problématiques et une vingtaine de types de messages invalides particulièrement défavorables. Néanmoins, certains de ces messages résultent de l'application d'un même opérateur de mutation à un message valide standard. Ainsi, nous avons synthétisé les différentes erreurs en quatre opérateurs de mutation. Nous avons également défini des opérateurs de mutation rendant un message valide problématique. Ces règles de mutation sont spécifiques aux protocoles fondés sur le protocole HTTP.

9.2.1 Opérateurs de mutations vers des messages invalides

En synthétisant les informations collectées depuis la RFC 4475, nous avons défini un ensemble d'opérateurs de mutation s'attaquant aux caractères, aux répétitions, aux valeurs entières et aux litté-

[1] http ://www.service.real.com/help/faq/security/bufferoverrun12192002.html

raux définis dans les grammaires des protocoles. L'application de ces opérateurs produit des mutants mettant en évidence des erreurs largement répandues, rencontrées lors de notre analyse de domaine.

Mutation des caractères La valeur d'un élément d'un message est généralement définie par la succession de caractères pris dans un ensemble de caractères variant selon l'élément. L'opérateur de mutation consiste à insérer un caractère absent de cet ensemble. Les caractères sont divisés en différentes classes, notamment, la classe des séparateurs qui regroupe des caractères utilisés pour marquer la fin d'un élément comme un espace ou un point-virgule. L'analyse syntaxique de cette classe des séparateurs s'avèrent particulièrement cruciale, car ces séparateurs marquent la frontière entre deux éléments successifs. L'insertion d'un caractère mutagène se fait au début et à la fin de l'élément considéré ainsi qu'à une position intermédiaire aléatoire, produisant trois mutants par caractère pris hors de l'ensemble. Le début et la fin d'un élément sont deux positions particulièrement problématiques pour l'analyse syntaxique puisqu'elles délimitent un élément.

Mutation des répétitions Un élément d'un message est défini à la fois par l'ensemble des caractères permis mais aussi par le nombre de répétitions permises pour les caractères de cet ensemble. Ce nombre de répétitions permet de définir indirectement une longueur minimale et maximale de chaîne de caractères. Par exemple, le code d'une réponse dans HTTP doit comprendre trois chiffres, dans le cas contraire, le message est invalide. L'opérateur de mutation consiste à utiliser des caractères valides mais avec un nombre de répétitions invalide. Un mutant est généré en dessous de la borne inférieure (si elle existe), un autre au dessus de la borne supérieure (si elle existe).

Mutation des entiers Le premier opérateur de mutation présenté mute des caractères en des caractères invalides. Un cas que nous distinguons est celui où l'ensemble des caractères valides est celui des chiffres. Nous faisons cette distinction car l'analyse syntaxique des nombres implémentée manuellement dans les applications que nous avons considérées diffèrent de l'analyse des caractères alphabétiques. Nous allons donc insérer des caractères alphabétiques dans ces nombres. Une autre mutation concernant les entiers consiste à valider le comportement de l'analyseur syntaxique aux bornes. Il est fréquent que les bornes d'un élément représentant un nombre soient définies de manière informelle dans le texte de la spécification. Généralement, ces nombres sont convertis directement depuis leur représentation sous forme de chaîne de caractères en entier sans plus de vérification.

Mutation des littéraux Nous parlons de littéraux lorsqu'un élément peut prendre la valeur d'une chaîne de caractères parmi plusieurs définies dans une énumération. L'opérateur de mutation que nous introduisons dérive des mutants de ces diverses valeurs. Typiquement, un mutant reprend une chaîne de caractères de cet ensemble à laquelle on adjoint un caractère. Cela permet de tester si l'analyseur syntaxique ne considère que les premiers caractères jusqu'à reconnaître une chaîne particulière.

9.2.2 Opérateurs de mutations vers des messages valides

Il est plus délicat de produire des messages valides potentiellement problématiques. Nous partons du principe que les analyseurs syntaxiques rencontrés gèrent de manière appropriée les messages usuels qui représentent une grande majorité des messages véhiculés. Nous nous attachons à tester le support de certaines subtilités présentes dans les différentes spécifications. La définition de ces différents opérateurs se fonde principalement sur notre expérience des erreurs rencontrées dans les différents supports protocolaires étudiés.

Caractères protégés Dans de nombreux cas, certains caractères spéciaux comme les séparateurs peuvent apparaître dans des éléments du message en étant *protégés* par un autre caractère dédié à cette protection. L'implémentation de cette protection, si présente, est souvent défectueuse. Nous introduisons donc ces caractères spéciaux protégés dans les éléments pouvant accueillir ces caractères. Ces caractères sont positionnés au début puis à la fin de l'élément considéré, puisque ces caractères protégés sont souvent des séparateurs, puis nous définissons une position intermédiaire aléatoire pour vérifier le comportement à l'intérieur de l'élément.

Longueur des éléments De nombreuses erreurs proviennent d'une mauvaise gestion de la longueur des tableaux ou plus généralement des tampons mémoires. En effet, les tableaux servant à accueillir les différents élément extraits du message sont souvent alloués statiquement et se cantonne à de faibles longueurs. Ainsi, on voit souvent apparaître des erreurs de débordement de tampons. Nous introduisons un opérateur de mutation permettant de valider le comportement de l'analyseur syntaxique par rapport à la longueur des chaînes de caractères considérées. Cet opérateur augmente la longueur d'un élément pour vérifier la bonne gestion de longs éléments.

Alternatives Nous l'avons vu précédemment, certains éléments du message sont définis à l'aide d'alternatives. Néanmoins, comme les protocoles que nous considérons sont extensibles, ces alternatives présentent généralement une clause d'extension. Une clause d'extension ajoute à un ensemble d'alternatives une alternative libre (mais qui respecte un certain format). Cela permet de normaliser un certain nombre de ces alternatives tout en permettant l'introduction de nouvelles alternatives lorsqu'une extension au protocole est spécifiée. L'opérateur de mutation crée des valeurs respectant la clause d'extension mais qui se rapprochent d'une des alternatives énumérées. Par exemple, une valeur créée reprend une des alternatives comme préfixe. Dans certains supports protocolaires que nous avons rencontrés, en effet, dés qu'une alternative est reconnue comme préfixe, l'analyse syntaxique considère qu'une alternative complète a été détectée. Ainsi, l'analyse syntaxique considère les caractères suivant comme faisant partie d'un autre élément, ce qui peut résulter sur une erreur.

9.3 Expérimentations

Notre objectif est d'évaluer l'amélioration de robustesse induite par l'utilisation de Zebu dans les analyseurs syntaxiques de protocoles réseaux textuels. Nous évaluons cette robustesse en comparant le comportement d'analyseurs syntaxiques fondés sur Zebu à divers analyseurs syntaxiques existants pour SIP et RTSP . Nous validons ce comportement en soumettant les analyseurs syntaxiques à la réception d'un ensemble de messages valides et invalides.

Pour SIP, nous comparons l'analyseur syntaxique fondé sur Zebu avec celui fourni par la bibliothèque de fonctions [Moi01] et celui présent dans le serveur mandataire, préalablement mentionné, SER [POJK03]. Pour RTSP, nous nous comparons à l'analyseur syntaxique du serveur de diffusion largement répandu, VLC [vlc] et celui fourni par la bibliothèque LiveMedia [liv]. Le tableau 9.1 expose les tailles des différents analyseurs syntaxiques considérés ainsi que de la grammaire ABNF de ces protocoles et enfin celles des spécifications Zebu. La spécification Zebu dans le cas de SIP compte douze fois moins de lignes de code que l'analyseur syntaxique de SER considéré comme une référence. Dans le cas de RTSP, le rapport est moins important mais principalement du fait de la simplicité des analyseurs syntaxiques écrits manuellement que nous avons considérés (nous le verrons par la suite). Nous voyons également sur ce tableau que les spécifications Zebu sont environ 50% plus longues que les spécifications ABNF correspondantes. Cet état de fait résulte d'une part que les

spécifications ABNF sont incomplètes dans le sens où elles font référence à d'autres spécifications et d'autre part à la définition de contraintes structurelles à l'intérieur de la spécification Zebu.

Protocole	Taille de l'ABNF	Taille de la spécification Zebu	Analyseur Syntaxique	Taille de l'analyseur syntaxique
SIP	≈ 700	1081	oSIP	11982
			SER	13277
RTSP	≈ 200	330	VLC	≈ 1200
			LiveMedia	≈ 1000

TAB. 9.1 – Tailles en lignes de code des grammaires des messages SIP et RTSP, et des analyseurs syntaxiques existants

9.3.1 Méthode d'expérimentation

Pour comparer la robustesse des analyseurs syntaxiques SIP et RTSP fondés sur Zebu à ceux écrits manuellement, nous avons créé des applications réduites, qui se focalisent sur les premières lignes et les en-têtes spécifiés comme obligatoire par les protocoles SIP et RTSP. Ces applications permettent la journalisation de différents éléments à des fins statistiques. Les applications fondées sur Zebu, `log-Zebu-SIP` et `log-Zebu-RTSP` respectivement pour SIP et RTSP, consistent seulement en quelques lignes de code C qui s'appuient sur les fonctions talons générées par le compilateur Zebu telles que celles présentées à la figure 7.6. L'analyseur syntaxique généré pour ces expérimentations effectue une analyse stricte du message avec validation totale de ces éléments.

La même application de journalisation a été implémentée dans un serveur mandataire SER. Ce serveur intègre deux niveaux de programmation. Un premier niveau est introduit par l'intermédiaire d'un langage dédié de configuration. Ce langage fournit un accès aux différents éléments du message par le biais de construction haut niveau qui reposent sur des fonctions implémentées dans le serveur. Le deuxième niveau est introduit par un système de modules qui permet de créer des fonctions ayant un accès direct au message, fonctions qui sont rendues disponibles au niveau du langage dédié de configuration. Nous avons choisi d'implémenter l'application, `log-SER` à l'aide du langage de configuration pour reposer sur les fonctions existantes de l'analyseur syntaxique.

Les autres applications `log-oSIP` utilisant oSIP, `log-VLC` utilisant VLC, et `log-LiveMedia` utilisant LiveMedia, sont écrites dans le langage C en utilisant les différentes fonctions fournies fondées sur les analyseurs syntaxiques pour extraire les données du messages.

Nous avons ensuite positionné des sondes dans les différentes applications au niveau du support protocolaire et à l'interface entre ce support protocolaire et la logique de l'application. Ces sondes nous permettent de contrôler le comportement du support protocolaire à la réception de messages. Ainsi, nous surveillons les différents indicateurs d'erreur qui sont censés se déclencher à la réception d'un message invalide.

9.3.2 Messages invalides

Les différentes expérimentations que nous avons menées nous ont démontré la viabilité de notre approche. En effet, que ce soit pour SIP ou pour RTSP comme le présente la figure 9.2, les analyseurs syntaxiques stricts générés par le compilateur Zebu détectent 100% des messages invalides de notre jeu de tests, mutés selon nos règles de mutation. Ce résultat est possible par la génération d'un analyseur complet strict. En utilisant une approche générative, nous escomptions atteindre une telle perfection dans la détection des messages invalides. Un taux de détection différent de 100% aurait démontré une faille dans notre compilateur.

		Messages invalides	Messages détectés	% messages detectés
SIP	`log-Zebu-SIP`	5976	5976	**100.0**%
	`log-oSIP`		1020	**17.1**%
	`log-SER`		1512	**25.3**%
RTSP	`log-Zebu-RTSP`	2730	2730	**100.0**%
	`log-VLC`		4	**0.1**%
	`log-LiveMedia`		748	**27.4**%

TAB. 9.2 – Couverture des analyseurs syntaxiques pour des messages SIP et RTSP invalides

Par opposition aux analyseurs syntaxiques fondés sur Zebu, les analyseurs syntaxiques développés manuellement ne détectent pas plus de 28% des messages invalides. Cette grande différence provient en partie d'une certaine flexibilité consentie par rapport à la spécification. Cette flexibilité est d'ailleurs souvent demandée dans les spécifications des protocoles que nous considérons. En effet, il est d'usage d'accepter des messages *à la limite* de la spécification pour ne pas isoler certaines applications, des clients notamment. Néanmoins, la flexibilité consentie éloigne les analyseurs syntaxiques de la spécification, cet éloignement est alors plus souvent subi que consenti. La conséquence est que cet éloignement introduit une mauvaise gestion de certaines classes de messages invalides.

Le plus étonnant est que l'analyseur syntaxique du serveur de diffusion VideoLan est extrêmement laxiste dans son analyse. En effet, seulement 0.1% des messages invalides transmis ont été détectés au niveau du support protocolaire. En lisant le code, on se rend compte que l'analyseur syntaxique est simpliste. Par exemple, il n'y a aucun contrôle sur la méthode d'une requête, l'analyseur syntaxique considère que la méthode équivaut à l'ensemble des caractères rencontrés jusqu'à un espace. Le problème est que le contrôle des erreurs est défaillant au niveau de la couche applicative. Ainsi, les messages reçus peuvent être affichés sans aucun contrôle, ce qui a permis l'exploitation de ce que l'on appelle les vulnérabilités de format de chaîne de caractères.

Les tests que nous avons menés en soumettant l'application à une série de messages invalides mutés selon les opérateurs définis dans la section 9.2.1 permettent également de valider le comportement du support d'erreurs, c'est-à-dire la détection, la collecte et le traitement des erreurs. En effet, nous appliquons successivement tous les opérateurs de mutation possibles à un élément du message puis nous passons à un autre élément du message. Or, le support d'erreurs est dépendant généralement de l'élément du message considéré. Ainsi, cette démarche séquentielle nous a permis de révéler une erreur dans une certaine version de SER qui conduit à une interruption totale du service. SER collecte les messages contenant un en-tête particulier erroné, l'en-tête `Via` utilisé pour le routage des messages SIP. Cette collecte s'avérait être mal implémentée, la réception d'un grand nombre de messages avec un en-tête `Via` erroné entraîne l'arrêt complet du serveur mandataire par une faute de segmentation. La téléphonie, à l'instar de la distribution d'eau ou d'électricité, est une commodité basale. L'arrêt inopiné d'un serveur de téléphonie isole un domaine, que ce soit une entreprise, un quartier, une ville, du reste du monde. Cette erreur, corrigé dans la version 0.9.6, a perduré durant près d'un an.

9.3.3 Messages valides

Nous avons appliqué les opérateurs de mutation présentés à la section 9.2.2 au même message de départ que précédemment. La figure 9.3 montre que l'analyseur syntaxique SIP fondé sur Zebu ne rejette aucun message. Nous pouvons donc dire que l'analyseur syntaxique est exact, il ne rejette aucun message invalide et accepte tous les messages valides. Par contre, les analyseurs syntaxiques développés manuellement pour le même protocole rejettent certains messages valides. Ces messages sont certes non-usuels mais le rejet de messages valides démontre le fait que la flexibilité introduite au niveau des analyseurs syntaxiques n'est pas réellement contrôlée. De manière plus problématique,

l'envoi de certaines requêtes fait échouer l'analyseur syntaxique de la bibliothèque oSIP.

		Messages valides	Messages rejetés	% messages rejetés
SIP	`log-Zebu-SIP`	549	0	**0.0**%
	`log-oSIP`		21	**3.9**%
	`log-SER`		2	**0.4**%

TAB. 9.3 – Couverture des analyseurs syntaxiques pour les messages SIP valides

Nous avons mené des expérimentations analogues pour le protocole RTSP avec les mêmes points de comparaison que précédemment. L'analyseur syntaxique de LiveMedia s'avère plus cohérent que les analyseurs syntaxiques pour SIP dans le sens où il ne rejette aucun message valide. Enfin, l'analyseur syntaxique de VLC est tellement laxiste que le fait qu'il ne rejette aucun message valide n'est pas réellement significatif. L'analyseur syntaxique fondé sur Zebu pour RTSP ne rejette aucun message valide.

9.4 Bilan

Les vérifications au niveau de la spécification Zebu que nous avons décrites à la section 7.4 permettent de garantir que la spécification est cohérente et peut donc être compilée.

Nous avons conduit dans cette section une évaluation de l'impact de l'utilisation de Zebu sur la robustesse du support protocolaire d'applications réseaux. Les résultats de nos expériences montrent que les analyseurs syntaxiques fondés sur Zebu sont exacts dans le sens où ils rejettent tous les messages invalides et acceptent tous les messages valides. Les analyseurs syntaxiques développés manuellement détectent seulement environ un quart des messages invalides. L'utilisation des données mal formées extraites de ces messages invalides laisse la porte ouverte à des attaques très simples à mettre en œuvre et qui peuvent permette d'interrompre le service fourni voire de prendre à distance le contrôle de la plate-forme matérielle.

Chapitre 10

Performances

Tous les messages, aussi bien en provenance qu'à destination d'une application réseau transitent par sa couche de support protocolaire. Cette couche représente l'interface entre le monde extérieur et l'application considérée. Par conséquent, une part significative des performances globales d'une application réseau dépend des performances de ce support protocolaire. Dans le chapitre précédent, nous avons évalué la robustesse du support protocolaire fondé sur Zebu. Ces évaluations ont montré que ce support est robuste car il respecte exactement la spécification. Néanmoins, le support protocolaire doit s'avérer robuste mais aussi très performant. Bien souvent, d'ailleurs, cette contrainte de performance prévaut sur celle de robustesse. Ainsi, l'impact d'une approche telle que la nôtre serait minime sans des performances au moins équivalentes à celles obtenues par un programme développé manuellement.

Nous avons évalué le coût à l'exécution engendré par Zebu en comparant les performances du code généré par ZebuYacc, la première version du compilateur, avec les performances de programmes développés manuellement. Pour que l'expérience soit significative, nous nous sommes comparés aux standard *de facto* pour chaque protocole considéré. Dans un second temps, nous avons évalué la mémoire consommée par le code généré par la chaîne de compilation μ-Zebu. Cette évaluation est une étude de faisabilité en conditions réelles du déploiement d'applications réseaux fondées sur Zebu sur des systèmes embarqués, et par voie de conséquence de l'utilisation de protocoles applicatifs textuels dans ce contexte. Dans la suite de ce chapitre, nous présentons les résultats de nos différentes expériences.

10.1 Évaluation de ZebuYacc

Cette section reprend les différentes expériences que nous avons menées sur le code généré par ZebuYacc pour évaluer les performances de ce code. Nous présentons, dans un premier temps, des micro-évaluations sur HTTP. Dans un second temps, nous exposons les résultats pour l'application fondée sur SIP présentée dans le chapitre précédent. Dans les deux cas, ces résultats sont obtenus sur des enregistrements de messages tirés d'échanges réels.

La procédure de test que nous suivons, quelque soit le protocole, est la suivante : un client extrait un par un les messages d'un enregistrement et envoie chaque message à un serveur qui exécute l'application considérée. Nous avons positionné des sondes au sein des différentes applications pour mesurer le temps de l'analyse syntaxique pour chacun des messages reçus. Le serveur utilisé pour ces expérimentations est un Pentium 4 (2.66GHz) et le client un Athlon Mobile (1.6GHz).

10.1.1 Micro-évaluations sur HTTP

Afin d'évaluer le temps de traitement d'un message HTTP par l'analyseur syntaxique inclus dans le code généré par ZebuYacc à partir de la spécification Zebu de HTTP, nous avons développé une application qui enregistre le nom de domaine contenu dans un en-tête particulier d'une requête, l'en-tête `HOST`. Cette spécification de HTTP compte 500 lignes et entraîne la génération de 2400 lignes de code C. L'application en elle-même compte seulement 27 lignes. Nous avons ensuite confronté notre implémentation fondée sur Zebu à deux autres implémentations : une fondée sur PADS et une sur le serveur Internet Apache.

L'implémentation fondée sur PADS repose sur le code généré à partir de la spécification de HTTP fournie avec le compilateur PADS. Cette spécification, de 2000 lignes environ, consiste en un ensemble ordonné (l'ordre de définition des règles étant significatifs) de types PADS classiques et d'expressions régulières pour exprimer les éléments du protocole difficilement exprimables en PADS. L'application en elle-même représente environ 50 lignes de code dont la majeure partie provient de l'initialisation de l'environnement d'exécution.

L'application fondée sur le serveur Internet Apache est implémentée à l'aide du fichier de configuration du serveur. Ce fichier permet de définir le comportement du serveur par l'utilisation de fonctions exposées par différents modules. Chaque module est dédiée à une tâche bien particulière, par exemple le support de l'authentification ou de XML. Le module qui nous intéresse est le module `mod_setenvif` permettant de définir des fonctions d'analyse syntaxique à partir d'expressions régulières pour chaque en-tête d'un message. L'application en elle-même représente un ajout de 9 lignes sur le fichier de configuration initial. Les 9 lignes ainsi ajoutées correspondent à des règles de filtrage des messages et ensuite d'extraction de l'information recherchée.

Comme nous pouvons le voir sur le tableau 10.1, l'analyse syntaxique avec validation complète d'un message par le code généré par ZebuYacc requiert 170,000 cycles en moyenne. Les évaluations effectuées sur l'application fondée sur PADS ont montré combien l'utilisation de cet outil est inadéquate dans un tel contexte avec 2,600,000 cycles en moyenne par message, soit un temps de traitement 1500 fois plus important. L'analyse syntaxique d'un message effectuée par Apache demande seulement 24,000 cycles, ce qui représente seulement 14% du temps requis par l'analyseur syntaxique généré par ZebuYacc. Néanmoins, ce résultat est à mettre en perspective avec le niveau d'analyse effectué. En effet, dans le cas d'Apache seul l'en-tête `HOST` est analysé complètement à l'aide d'une expression régulière fournie, pour les autres, un pointeur vers leur position de départ est initialisé sans autre analyse. Ainsi, l'analyse syntaxique effectuée par le code généré par ZebuYacc est plus lente car plus complète.

ZebuYacc	PADS	Apache
170 000	2 600 000	24 000

TAB. 10.1 – Temps pour l'analyse syntaxique d'un message HTTP (en nombre de cycles)

10.1.2 Évaluations sur SIP

Pour évaluer les performances de l'analyseur syntaxique généré par ZebuYacc à partir de la spécification Zebu de SIP, nous avons repris la même application que celle utilisée pour l'évaluation de robustesse vue dans le chapitre 9. Cette application consiste à enregistrer les domaines d'émission des appels reçus. Pour cela, la logique de l'application extrait l'information correspondante de l'en-tête `FROM` des requêtes `INVITE` reçues. Pour évaluer la robustesse, nous avions *construit* un ensemble

de messages invalides ou potentiellement problématiques à partir d'une RFC présentant de tels messages. Pour les performances, nous nous sommes placés dans un contexte plus proche des conditions réelles d'utilisation de notre application. Pour cela, nous avons collecté un enregistrement de messages SIP réels échangés au sein de l'Université de Bordeaux. Cet enregistrement représente un jour complet de trafic de téléphonie sur IP comptant près de 3000 messages. L'application, qui extrait le nom d'hôte de l'URI de l'en-tête `From` pour les requêtes `INVITE`, illustre le cas où une application telle qu'un système de détection d'intrusion nécessite l'extraction d'un sous-champs d'un message. Les implémentations fondées sur Zebu, `Zebu-full` et `Zebu-minimal`, comptent toutes les deux 28 lignes et reprennent la structure définie à la figure 8.2. En effet, ces deux applications ne varient pas au niveau de la logique applicative mais seulement au niveau du support généré par ZebuYacc utilisé. Ce support varie par la validation du message. Si `Zebu-full` effectue une validation complète du message, `Zebu-minimal` valide seulement les élément du message annotés au niveau de la spécification comme étant utilisés par l'application. En guise de points de comparaison, nous avons développé trois applications reposant sur d'autres infrastructures, deux fondées sur SER et une sur oSIP.

Le serveur SER intègre deux niveaux de programmabilité, nous avons donc implémenté une instance de l'application pour chacun de ces niveaux. Le premier niveau, à l'aide d'un langage à script, repose sur les fonctionnalités existantes dans le serveur pour router un message. Ce langage est dédié au routage et ne permet donc pas certaines opérations de filtrage de motifs. Néanmoins, SER fournit un échappatoire en permettant l'invocation d'un script *shell* arbitraire. Cette approche, cependant, présente un inconvénient majeur : la pénalité en termes de performance induite par le changement de contexte (à laquelle il faut ajouter les potentiels problèmes d'erreur dans ce script). Pour pallier ce problème, il existe dans SER un deuxième niveau de programmabilité, une infrastructure d'extension permettant l'intégration de nouveaux modules (à l'instar du serveur Apache). Ces modules ont un accès complet aux messages et ne sont pas limités dans leurs fonctionnalités. Ainsi, il existe des modules pour l'authentification ou l'interconnexion avec une base de données. Bien qu'efficace, cette approche demande au programmeur d'écrire du code C relativement bas niveau, ce code doit également respecter l'ensemble des contraintes, notamment en termes de prototype des fonctions, définies par l'infrastructure d'extension. Ainsi, la première version `SER-exec` est écrite en utilisant le langage de configuration et repose sur l'invocation d'un script *shell* pour extraire l'information de nom d'hôte, ce qui représente 12 lignes de code. La seconde version `SER-module` est implémentée comme un module SER pour obtenir un accès complet aux structures internes du serveur. Ce module, en lui-même, compte 150 lignes et requiert l'ajout de 22 lignes au fichier de configuration. Le dernier point de comparaison repose sur la pile oSIP, `oSIP`, consiste en 40 lignes de code C qui configurent la pile et appelle les différentes fonctions que la pile fournit.

Nous avons soumis ces différentes implémentations à la réception des messages extraits de l'enregistrement d'échanges réels. Les résultats de cette évaluation sont représentés sur le tableau 10.2 en les rapportant aux performances de `Zebu-minimal`

L'analyse syntaxique de SER repose sur une stratégie à deux niveaux équivalentes à celle que nous avons adoptée, un analyseur syntaxique gros grain et un ensemble d'analyseurs syntaxiques spécialisés par en-tête. L'analyse syntaxique effectuée par `SER-module` s'avère particulièrement efficace, jusqu'à trois fois plus rapide que `Zebu-minimal` dans le cas de requêtes, l'information à extraire étant disponible dans les structures de données internes à SER. Cependant, l'analyse syntaxique des réponses est 50% plus lente que celle effectuée par `Zebu-minimal`. La raison est que SER est dédié au routage de messages et requiert un temps incompressible pour l'analyse syntaxique de certains éléments du message cruciaux pour le routage, tels que l'en-tête `Via`, mais inutiles pour notre application. Ainsi, Zebu peut fournir de meilleures performances par une génération de code dédiée à

		SER-module		SER-exec		oSIP		Zebu-full		Zebu-minimal	
		Cycles	**Ratio**	**Cycles**	**Ratio**	**Cycles**	**Ratio**	**Cycles**	**Ratio**	**Cycles**	**Ratio**
Requête	ACK	25 460	32%	22 690	29%	377 298	481%	81 796	104%	78 406	100%
	BYE	22 540	42%	36 608	69%	522 156	984%	77 468	146%	53 060	100%
	CANCEL	22 732	46%	52 996	107%	336 036	676%	84 512	170%	49 680	100%
	INVITE	34 868	43%	7 593 550	9 438%	525 654	653%	81 670	102%	80 456	100%
	OPTIONS	25 872	36%	24 176	34%	339 016	478%	76 144	107%	70 908	100%
	REGISTER	31 016	50%	31 096	50%	465 978	751%	65 672	106%	62 028	100%
Réponse		35 608	151%	35 716	152%	505 312	2 150%	31 806	135%	23 508	100%

TAB. 10.2 – Performance de l'application de statistiques sur des messages SIP réel (temps en cycles, ratio par rapport à `Zebu-minimal`)

une application particulière, tout en garantissant sa robustesse, qui peut être compromise par l'utilisation de modules de SER. En effet, les modules de SER jouissent d'un accès illimité aux structures de données internes du serveur, ce qui peut compromettre, en cas d'erreur, sa robustesse.

L'analyse syntaxique pratiquée par `SER-exec` est approximativement aussi efficace que celle effectuée par `SER-module` pour les messages de type autre que `INVITE`. Pour ceux de type `INVITE`, la différence est importante, `SER-exec` étant 217 fois plus lent, principalement car un processus *shell* est créé. Malgré ses mauvaises performances dans ce cas, l'utilisation du langage de configuration de SER reste une approche valide car elle combine facilité de programmation et robustesse.

L'analyse syntaxique effectuée par `oSIP` est entre 5 et 10 fois plus lente que l'analyse syntaxique de `Zebu-minimal` pour les requêtes et plus de 21 fois pour les réponses. Dans ces deux cas, la pile oSIP analyse les six en-têtes SIP obligatoires, auxquels s'ajoutent deux en-têtes spécifiques aux messages de type `REGISTER`, et stocke, pour chacun de ces en-têtes, un pointeur vers sa position de départ. Lorsque l'application requiert certaines informations concernant les en-têtes d'une requête `INVITE`, la pile oSIP copie ces sous-champs dans des structures de données accessibles par l'application.

Ces différentes évaluations ont montré que le code généré par ZebuYacc qui respectent strictement la spécification ABNF du protocole présente de bonnes performances parmi les approches fournissant des garanties de sûreté et de robustesse.

10.2 Évaluation de la chaîne de compilation optimisée μ-Zebu

La section précédente a montré que les applications fondées sur Zebu et ZebuYacc représentaient une alternative viable à celles fondées sur des infrastructures existantes quant à la combinaison entre la robustesse et les performances. Nous l'avons évoqué précédemment, la mémoire limitée des systèmes embarqués s'accommode mal de la prise en charge de protocoles réseaux complexes aux analyseurs syntaxiques gourmands en ressources. Pour cette raison nous avons développé la chaîne de compilation μ-Zebu visant à permettre l'adoption de protocoles applicatifs textuels dans le contexte des systèmes embarqués. μ-Zebu intègre, notamment, des évolutions de ZebuYacc en termes de génération de code. Dans cette section, nous présentons l'évaluation de μ-Zebu dans des conditions réelles, une application de surveillance de bâtiment déployée dans notre infrastructure pilote. Cette application permet à un téléphone SIP de contrôler et recevoir des images d'une caméra IP, de marque Axis en l'occurrence. La caméra est un système clos qui gère seulement le protocole RTSP quant à la négociation de paramètres des sessions multimédias (audio et vidéo) et accepte seulement le protocole HTTP pour la commande d'opérations telles que le zoom ou le mouvement. Cependant, notre infrastructure pilote supporte seulement SIP. Par conséquent, nous avons créé une passerelle pour convertir les messages SIP en RTSP ou HTTP, et inversement, suivant l'action requise. Cette passerelle implique

l'analyse syntaxique et la construction de messages SIP, HTTP et RTSP. Nos expérimentations se focalisent sur l'implémentation de cette passerelle en tant que système embarqué, comme cela serait nécessaire dans un contexte de surveillance de bâtiment où l'on souhaite limiter les coûts matériels.

Pour nos expérimentations, nous avons installé la passerelle sur des systèmes embarqués de deux natures. Pour comparer µ-Zebu avec des systèmes présentant des ressources conséquentes, nous avons implémenté la passerelle sur une carte CPUAT91, commercialisée par Eukréa. Cette carte repose sur un processeur ARM9 cadencé à 200Mhz et comporte 32Mo de SDRAM, 8Mo de mémoire flash, 1Ko d'EEPROM. Le système d'exploitation que nous y avons installé est un Linux avec un noyau dans sa version 2.6.20. Pour une évaluation dans un contexte plus contraint, nous avons utilisé un kit d'évaluation et de développement Atmel EVK1100 reposant sur un micro-contrôleur AVR32 avec 512Ko de flash, 32Mo de RAM sur un circuit séparé, et pas de MMU. Le client de téléphonie SIP que nous utilisons pour contrôler la caméra et afficher les images qu'elle renvoie est Linphone [Lin]. Actuellement, seule la passerelle est implémentée à l'aide de µ-Zebu.

10.2.1 Structure de l'application

La figure 10.1 illustre les différents composants mis à contribution dans le cadre de notre application. Un client SIP standard envoie un message de type `INVITE` à la passerelle SIP pour recevoir le flux vidéo capturé par la caméra. La passerelle extrait les informations appropriées de ce message, telles que l'identité de l'appelant, pour les intégrer dans un message RTSP que la passerelle transmet à la caméra. Une fois la communication multimédia établie, la passerelle n'est plus concernée, et le flux vidéo est transmis directement de la caméra au client SIP par le protocole RTP.

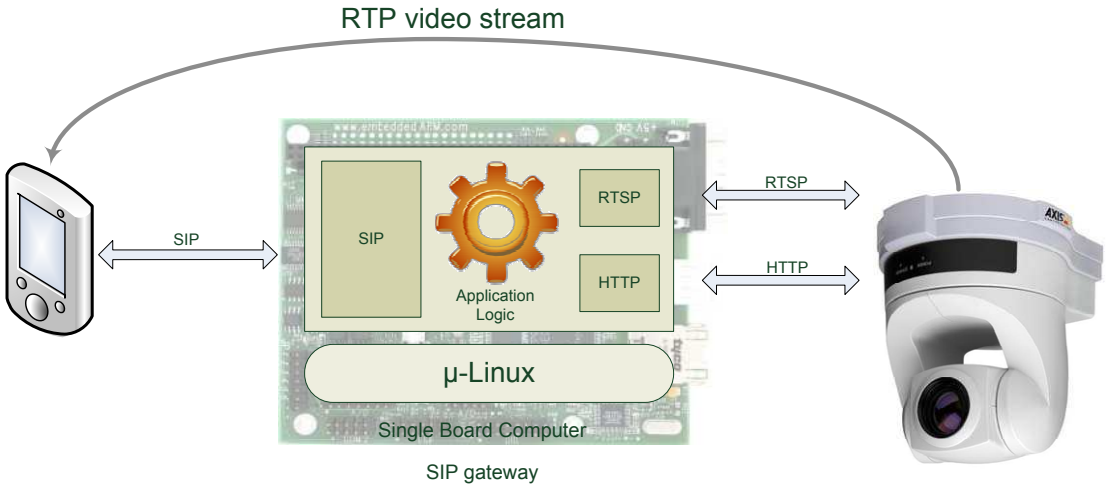

FIG. 10.1 – Passerelle SIP pour la caméra Axis

Pour contrôler les différentes fonctionnalités fournies par la caméra, le zoom et les différents mouvements, nous avons étendu le protocole SIP avec de nouvelles méthodes et de nouveaux en-têtes représentant chacune de ces fonctionnalités. Par exemple, pour zoomer, nous avons introduit une nouvelle méthode SIP `ZOOM` et un nouvel en-tête `Zone` qui décrit la zone sur laquelle zoomer, et le facteur de zoom à utiliser par la caméra. La figure 10.2 montre les lignes à rajouter dans la spécification Zebu de SIP pour prendre en charge cette extension. Parallèlement, nous avons étendu le client Linphone pour supporter ces différentes extensions. Ainsi, un assistant personnel qui exécute ce client

peut recevoir les images capturées par la caméra et piloter ses mouvements à distance. Quand la passerelle reçoit une requête d'un type nouvellement défini par une extension, elle extrait l'information requise (par exemple l'en-tête `Zone` pour une requête de type `Zoom` pour construire la requête HTTP correspondante et l'envoyer à la caméra. La réponse HTTP renvoyée par la caméra est capturée par la passerelle qui la convertit en une réponse SIP.

```
Method =/ ZOOMm
ZOOMm  = %x5A.4F.4F.4D ; ZOOM in caps
header Zone = *1(Hor SP) *1(Vert SP)  *1("-") 1*DIGIT
Hor  = ( "Left" / "Right" ) "=" *1("-") 1*3DIGIT
Vert = ( "Up" / "Down" ) "=" *1("-") 1*3DIGIT
```

FIG. 10.2 – Spécification Zebu de l'extension Zoom

Nous avons développé quatre versions de la passerelle à implémenter sur la carte CPUAT91 reposant sur un ARM9 : deux utilisant le code généré par μ-Zebu avec la partie arrière Linux et deux fondées sur un support protocolaire développé manuellement. Pour ce développement manuel, nous avons utilisé les bibliothèques eXosip [exo03] développée par AntiSIP[ANT] et Sofia-SIP [sof06] développée par un centre de recherche de Nokia et déployée sur certains de leurs produits. Nous avons également implémenté deux versions de la passerelle à déployer sur la carte EVK1100 à base d'AVR32 en utilisant la partie arrière FreeRTOS. Pour chacune de ces architectures, une version μ-Zebu effectue une validation complète du message tandis que l'autre n'en effectue aucune. Dans les différentes implémentations, la couche de logique applicative de la passerelle, reste identique du point de vue des fonctionnalités, en extrayant les informations appropriées des messages SIP puis en envoyant à la caméra les messages RTSP et HTTP correspondants et inversement.

10.2.2 Besoins en mémoire statique

Dans un premier temps, nous considérons les besoins en mémoire statique de la passerelle compilée pour la carte reposant sur un ARM9. Ces besoins englobent la taille du système Linux sans aucune application installée, la taille du support protocolaire compilé et la taille de la couche de logique applicative de la passerelle. Sans aucune application et en utilisant un système de fichier racine minimal, Linux utilise 1.8Mo. Le tableau 10.3 reprend la taille du support protocolaire compilé pour chaque version. On peut y voir que la taille du code compilé généré par μ-Zebu effectuant une validation complète représente seulement 35% du code équivalent pour eXosip et 11% du code équivalent pour Sofia-SIP. L'écart s'avère encore plus conséquent pour la version sans validation, représentant seulement 13% du code équivalent pour eXosip et 4% pour Sofia-SIP. Enfin, il est intéressant de noter que quelque soit le support protocolaire utilisé, la couche de logique applicative ne varie quasiment pas en taille avec 16.7 Ko, 17.7Ko et 17.8Ko pour, respectivement, Sofia-SIP, eXosip et μ-Zebu.

μ-Zebu		eXosip	Sofia-SIP
Validation complète	Pas de validation		
139	53	400	1 300

TAB. 10.3 – Taille, en kilo-octets, du support protocolaire compilé pour ARM9

Nous considérons maintenant les besoins en mémoire statique de la passerelle quand elle est compilée pour la carte EVK1100 fondée sur un AVR32. Pour nos expérimentations, nous avons utilisé μ-ZebuLink avec la partie arrière FreeRTOS pour générer une application complète intégrée dans le

système. Le tableau 10.4 montre que les besoins en ROM de la passerelle logent dans les 512Ko de la carte. Les besoins statiques en RAM occupent seulement une infime partie des 32Mo disponibles.

	ROM	RAM	Total
Validation complète	234.7	64	298.7
Pas de validation	150.6	64	214.6

TAB. 10.4 – Ventilation entre la ROM et la RAM de la taille du code de la passerelle complète sur la carte EVK1100 (en kilo-octets)

10.2.3 Besoins en mémoire dynamique

Dans le but d'évaluer la consommation mémoire à l'exécution des différentes versions de notre passerelle dans des conditions équitables, nous considérons une séquence d'actions figée. En premier lieu, le client envoie une requête de type `INVITE` à la passerelle pour recevoir le flux vidéo capturé par la caméra. Une fois la communication établie, le client envoie une séquence de requêtes `ZOOM` à la passerelle pour modifier le facteur de zoom de la caméra et la zone considérée. Successivement, le client demande un mouvement de 5 degrés sur la gauche avec un facteur de zoom de 5 puis un mouvement de 5 degrés sur la gauche avec un facteur de zoom négatif de -5 et enfin un mouvement de 30 degrés vers le haut avec un facteur 10 et enfin 30 degrés vers le bas avec un facteur -10. Pour clore la communication, le client envoie ensuite à la passerelle une requête `BYE`. Pour mesurer la consommation de mémoire dynamique engendrée par cette séquence d'actions, nous avons utilisé l'outil `exmap-console` [Exm]. `Exmap-console` s'appuie sur un module du noyau Linux pour fournir des informations très fines de consommation mémoire, la consommation mémoire pour chaque processus, et pour chaque processus la consommation pour chaque fichier objet.

Le tableau 10.5 représente la mémoire maximale requise pour exécuter la séquence `INVITE`/`ZOOM`-/`BYE` pour chaque version de la passerelle. Cette consommation mémoire dépend seulement du message couramment traité, aucun état n'étant maintenu au niveau de la passerelle. La colonne *Mémoire virtuelle* indique l'espace d'adressage alloué pour la passerelle ; la colonne *Mémoire physique* indique la quantité de mémoire physique réellement utilisé par la passerelle. Nous avons implémenté la passerelle en la fondant sur une couche de support protocolaire générée par ZebuYacc, nous avons représenté la consommation mémoire de cette implémentation à la première ligne du tableau 10.5. Comparé à ces résultats obtenus par la première version du compilateur, ZebuYacc, sans optimisation, on voit que les optimisations introduites dans la chaîne de compilation μ-Zebu réduisent la mémoire virtuelle requise de 14%. Ces optimisations réduisent la mémoire physique réellement consommée de 27% à 37%. Par ailleurs, les implémentations fondées sur μ-Zebu nécessitent seulement 40% de la mémoire virtuelle requise pour les implémentations fondées sur des supports protocolaires développés manuellement. Concernant la mémoire physique, l'impact de la validation complète sur l'implémentation fondée sur μ-Zebu sans validation représente seulement un ajout de 15%. La consommation en mémoire physique des implémentations fondées sur μ-Zebu avec validation et sans aucune validation représente respectivement 59% et 51% de celle consommée par l'implémentation eXosip, et, 41% et 35% de celle consommée par l'implémentation Sofia-SIP.

Néanmoins, il est important de noter que cette réduction de l'occupation mémoire ne s'est pas faite au détriment du temps d'exécution. Le tableau 10.6 montre la durée de l'analyse syntaxique d'une requête SIP de type `INVITE` initiant la communication avec la passerelle. Ce message représente le pire cas pour l'analyse syntaxique car il contient un ensemble d'éléments complexes nécessaires à l'établissement de la communication. À titre de comparaison, cette requête englobe treize en-têtes quand les autres messages échangés n'en contiennent que sept chacun. Avec une validation complète

		Mémoire virtuelle	Mémoire physique
ZebuYacc		2 292	868
μ-Zebu	Validation complète	1 972	632
	Pas de validation	1 972	548
eXosip		4 488	1 068
Sofia-SIP		5 056	1 544

TAB. 10.5 – Empreinte mémoire des processus en kilo-octets

activée, ce qui implique dans le pire cas pour les implémentations fondées sur μ-Zebu, l'analyse syntaxique s'effectue en 3500 μ-secondes, soit 10% de plus que pour l'analyse syntaxique effectuée par eXosip mais deux fois plus lentement que l'analyse syntaxique par Sofia-SIP dont la performance est un point fort. Par contre, sans validation, l'analyse syntaxique fondée sur μ-Zebu concurrence Sofia-SIP et s'avère deux fois plus rapide que l'analyse syntaxique effectuée par eXosip.

μ-Zebu		eXosip	Sofia-SIP
Validation complète	Pas de validation		
3 513	1 517	3 849	1 489

TAB. 10.6 – Temps pour l'analyse syntaxique d'un message SIP dans le pire cas (temps μ-secondes, moyenne sur 10 tests)

Nous détaillons maintenant la consommation mémoire physique et virtuelle de chacune des couches de la passerelle. En effet, la passerelle peut être subdivisée en trois couches distinctes : la *couche du support d'exécution* comprend les fonctions de commodité utilisées par l'application, des `sockets` aux tables de hachage. La *couche de support protocolaire* gère l'analyse syntaxique et la création de messages, c'est la couche générée par notre compilateur. Enfin, la *couche de logique de l'application* repose sur les deux couches précédentes pour implémenter la fonctionnalité de la passerelle. Les couches de support d'exécution et de support protocolaire sont dépendantes de chaque implémentation. La couche de logique applicative est globalement identique pour toutes les implémentations. Le tableau 10.7 montre la consommation mémoire en pic pour la séquence de messages précédemment évoquée pour les implémentations fondées sur eXosip, Sofia-SIP et μ-Zebu avec validation complète. La mémoire utilisée par le support d'exécution varie légèrement selon les fonctionnalités fournies et le comportement de la couche de support protocolaire, la mémoire utilisée par la couche de logique applicative ne variant pas.

Si nous nous intéressons à la couche de support protocolaire, nous voyons que l'implémentation fondée sur Sofia-SIP consomme le plus de mémoire, tout en étant la plus rapide des quatre implémentations. La couche de support protocolaire utilise plus de 1.1Mo de mémoire virtuelle pour 640Ko de mémoire physique, soit plus de 85% de la consommation mémoire totale. En fait, Sofia-SIP dans son analyse syntaxique complète d'un message crée de nombreuses structures de données. Chaque en-tête est copié dans un tampon mémoire, il est ensuite analysé syntaxiquement et chaque élément se voit stocker dans une nouvelle structure de données dédiée. Les grands besoins en mémoire de Sofia-SIP s'expliquent aussi par sa stratégie de modification de message très flexible mais très gourmande en mémoire. Les modifications de message sont effectuées grâce à un concept générique d'étiquettes (*tags*), qui permet d'effectuer des modification de diverses manières sur la vue structurée du message ou directement sur les données brutes.

La couche de support protocolaire de l'implémentation fondée sur eXosip consomme beaucoup moins de mémoire que son équivalent fondé sur Sofia-SIP, mais cette consommation représente tout de même 72% de sa consommation totale. EXosip repose sur la librairie oSIP dont nous avons évalué la robustesse dans le chapitre précédent, offrant seulement des fonctions de plus haut niveau. Ainsi,

	Application fondée sur μ-Zebu (validation complète)		Application fondée sur Sofia-SIP				Application fondée sur eXosip			
	Mémoire		**Mémoire**		**Ratio**		**Mémoire**		**Ratio**	
	Virt.	**Phys.**	**Virt.**	**Phys.**	**Virt.**	**Phys.**	**Virt.**	**Phys.**	**Virt.**	**Phys.**
Support d'exécution	96	60	160	92	167 %	153 %	260	124	271 %	207 %
Support protocolaire	188	44	1116	640	594 %	1 455 %	404	360	215 %	818 %
Logique de l'application	16	16	16	16	100 %	100 %	16	16	100 %	100 %

TAB. 10.7 – Consommation mémoire pour chaque couche (mémoire en kilo-octets, ratio par rapport à l'application fondée sur μ-Zebu)

eXosip fournit des structures de données de haut niveau masquant les complexités du modèle transactionnel de SIP. Cependant, eXosip réutilise les structures de données de description de message définies par oSIP. Essentiellement, la stratégie d'analyse syntaxique d'eXosip est équivalente à celle de Sofia-SIP. Par contre, les modifications de message s'effectuent en allouant une nouvelle structure de données représentant l'élément à modifier. Cette nouvelle structure est ensuite initialisée puis les pointeurs représentant l'élément du message associé sont mis à jour pour pointer vers cette nouvelle structure.

L'implémentation fondée sur μ-Zebu avec validation complète utilise moins de la moitié de la mémoire virtuelle requise par eXosip et moins d'un huitième de sa mémoire physique.

10.3 Bilan

Dans ce chapitre, nous avons en premier lieu confronté le support protocolaire généré par ZebuYacc à des enregistrements de messages, HTTP puis SIP, tirés d'échanges réels. Ainsi, nous avons vu que ce code généré par ZebuYacc, à niveau de robustesse équivalent, s'avère compétitif en termes de temps d'exécution.

Ensuite, nous nous sommes attachés à démontrer la faisabilité de l'utilisation d'une telle approche dans le contexte des systèmes embarqués. Grâce aux optimisations mémoires introduites dans la chaîne de compilation μ-Zebu, nous avons obtenu des résultats satisfaisants, permettant d'envisager l'utilisation de protocoles applicatifs textuels dans les systèmes embarqués.

Chapitre 11

Conclusion

Le développement des infrastructures de communication et la démocratisation du réseau Internet ont provoqué l'avènement des applications réseaux. Une application réseau se divise en deux couches, une couche de support protocolaire et une couche de logique applicative. La couche de support protocolaire gère la manipulation des messages, aussi bien l'analyse syntaxique des messages reçus que la modification ou la construction des messages à transmettre. Cette couche représente l'interface entre l'application à proprement parler, c'est-à-dire la logique applicative, et le monde extérieur. Toutefois, bien que cette couche soit un composant critique de l'application tant en termes de performance que de robustesse, son processus de développement est resté rudimentaire. Ce processus de développement requiert un haut niveau d'expertise dans divers domaines pour garantir les propriétés de cette couche, ce niveau d'expertise restreint l'*accessibilité au développement* à peu de personnes.

Nous avons présenté dans ce document une approche d'aide au développement du support protocolaire d'applications réseaux. Cette approche repose sur l'introduction d'un langage dédié à ce domaine nommé Zebu. Ce langage, fondé sur le formalisme utilisé pour la spécification de protocoles, permet le développement rapide d'une couche de support protocolaire, robuste et efficace, adaptée aux besoins d'une logique applicative particulière. Le traitement d'une spécification Zebu commence par son analyse afin de déceler d'éventuelles incohérences. La couche de support protocolaire pour la manipulation de messages est ensuite générée automatiquement. Des outils attenants au compilateur permettent une intégration complète du code généré dans une infrastructure existante.

Ce chapitre dresse un bilan des différentes contributions de cette thèse, puis recense un ensemble de perspectives de recherche potentielles.

11.1 Contributions

Les contributions de cette thèse peuvent être classifiées en deux volets. Dans un premier temps, nous montrons la faisabilité et l'intérêt d'une approche langage dans des contextes fortement contraints tant en termes de robustesse que de performance. Nous présentons le processus complet de conception d'un langage dédié à la spécification du support protocolaire d'applications réseaux : depuis les analyses préalables jusqu'à l'implémentation du langage. Dans un second temps, nous validons notre approche par l'évaluation du langage Zebu.

11.1.1 Analyses de domaine et de famille de programmes

Nous avons effectué une analyse complète du domaine des applications réseaux fondées sur des protocoles applicatifs textuels. Nous avons complété l'analyse de domaine par une analyse de la famille de ces applications. Les résultats de ces analyses nous ont permis d'identifier les paramètres d'entrée à la conception d'un langage dédié. Ces paramètres définissent les objets de base du domaine avec leurs opérations, relations et contraintes.

11.1.2 Zebu

À partir du cahier des charges défini par les analyses de domaine et de familles de programmes, nous avons conçu un langage dédié, nommé Zebu, pour spécifier le support protocolaire nécessaire à une logique d'application particulière par l'annotation des éléments requis au niveau de la logique applicative. Zebu repose sur le formalisme utilisé pour la spécification de protocoles, permettant comme nous l'avons montré de réduire l'effort de programmation tout en introduisant des abstractions offrant des opportunités de vérification de cohérence. Cette réduction de l'effort de programmation réduit par définition les sources potentielles d'erreur. De plus, Zebu permet d'étendre, de manière simple et sûre, le protocole sous-tendant une logique applicative pour y intégrer des aspects applicatifs très spécifiques.

Enfin, nous avons écrit des spécifications Zebu pour divers protocoles soutenant diverses logiques applicatives. Cette double diversité montre que l'expressivité du langage que nous proposons est suffisante.

11.1.3 ZebuYacc et μ-Zebu

Nous avons développé un compilateur pour Zebu, nommé ZebuYacc, générant du code C. Ce compilateur produit automatiquement à partir d'une spécification, une couche de support protocolaire complète. Une couche de support protocolaire complète comprend un analyseur syntaxique de messages fiable et performant, des structures de données représentant un message et ses différents éléments, un moyen d'accéder à ces structures de données et un moyen de modifier un message.

μ-Zebu est une chaîne de compilation complète dans laquelle nous avons intégré des techniques de réduction de consommation mémoire, principalement pour la cible des systèmes embarqués. À l'intérieur de cette chaîne de compilation, μ-ZebuYacc est une évolution de ZebuYacc prenant en compte des problématiques de consommation mémoire chères aux systèmes embarqués. μ-ZebuGen génère le code dédié à la création de message, et μ-ZebuLink permet le développement rapide d'application pour divers systèmes d'exploitation cibles.

Nous avons également décrit le processus de développement d'une application réseau reposant sur notre approche. L'accès au message par le biais de fonctions utilitaires respectant la vue du message souhaitée rend naturel le développement de la logique applicative.

11.1.4 Robustesse et efficacité

Nous avons conduit une évaluation de l'impact de l'utilisation de Zebu sur la robustesse et l'efficacité d'une application réseau. Les résultats des expériences que nous avons menées sur des applications représentatives montrent que le support protocolaire fondé sur Zebu détecte 100% des messages erronés reçus quand les implémentations manuelles détectent seulement 25% d'entre eux au mieux. De plus, le support protocolaire généré s'avère, à robustesse équivalente, aussi efficace que du support protocolaire développé manuellement.

L'évaluation de μ-Zebu, dans des conditions réelles, a également démontré la faisabilité de l'introduction de protocoles applicatifs textuels dans le monde des systèmes embarqués. La condition *sine qua non* est l'utilisation d'outils adaptés, notamment pour l'intégration d'une nouvelle application dans une infrastructure existante.

11.2 Perspectives

Les travaux présentés dans ce document décrivent une nouvelle approche pour le développement du support protocolaire d'applications réseaux. Cette approche s'inscrit dans la lignée d'autres approches fondées sur l'utilisation de langage dédiés pour le développement des différents composants d'un système d'exploitation [MCM+00, MLBS03, MRC+00].

Dans le chapitre 7, nous avons évoqué les différentes vérifications effectuées statiquement au niveau de la spécification. Par contre, nous avons validé le comportement à l'exécution du support protocolaire généré par un dérivé de la technique d'analyse appelé *analyse de mutations*. Une de nos prochaines directions de recherche consiste à produire une preuve formelle de correction de l'implémentation à l'aide d'outils tels que Coq [Tea].

Une autre perspective intéressante s'inspire du travail de Dawson Engler [ME04]. Une spécification Zebu peut être utilisée comme origine à la génération d'un outil de supervision et validation du support protocolaire d'applications réseaux développés manuellement. Cette certification du code par rapport à la spécification du protocole permettrait de prévenir un grand nombre des attaques que le support protocolaire subit. En cas d'erreur, un diagnostic explicite est retourné pour permettre la correction du programme.

Les travaux autour de Zebu prennent leur source dans un contexte plus global autour de langages dédiés modulaires. Pour paraphraser Rubén Prieto-Diaz dans son introduction à l'analyse de domaine [PD90], un large domaine est un regroupement de domaines plus étroits inter-connectés généralement structurés en un graphe dirigé. Ainsi, il serait pertinent de prendre des langages dédiés à des domaines très restreints pour les composer, comme autant de briques, afin de former un nouveau langage. Bien que les langages enchâsses semblent avoir pris une longueur d'avance grâce à des outils tels que Silver [VWS07, VWKSB07] ou JastAdd[EH07a], la combinaison de langages dédiés semblent une opportunité intéressante. En effet, hormis la création de nouveaux langages, cette modularisation permet de définir des familles de langages dont tous les membres, à l'instar des familles de programmes, présentent d'importants points communs mais se caractérisent également par leur variation. Dans notre cas, le langage que nous avons présenté dans ce document ne traite que de la couche basse d'une application que nous avons dissociée explicitement de la couche de logique applicative. L'implémentation de la logique applicative pourrait reposer sur un autre langage dédié intégrant des traits définis par une spécification Zebu.

Bibliographie

[ADH+89] H. AGRAWAL, R. DEMILLO, R. HATHAWAY, Wm. HSU, W. HSU, E. KRAUSER, R.J. MARTIN, A. MATHUR, et E. SPAFFORD. « Design of Mutant Operators for the C Programming Language ». Rapport Technique SERC-TR-41-P, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana, 1989.

[AJ74] A. V. AHO et S. C. JOHNSON. « LR Parsing ». *ACM Comput. Surv.*, 6(2) :99–124, 1974.

[ANT] « ANTISIP – and VoIP becomes easy. ». http ://www.antisip.com/.

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

[Ara89] G. ARANGO. « Domain analysis : from art form to engineering discipline ». Dans *IWSSD '89 : Proceedings of the 5th international workshop on Software specification and design*, pages 152–159. ACM, 1989.

[AS03] K. ARABSHIAN et H. SCHULZRINNE. « A SIP-based medical event monitoring system ». Dans *Proceedings of the 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry, 2003. Healthcom 2003*, pages 66–70, 2003.

[BA05] A. BEN-AMRAM. « The Church-Turing thesis and its look-alikes ». *SIGACT News*, 36(3) :113–114, 2005.

[BBW+07] N. BORISOV, D. J. BRUMLEY, H. J. WANG, J. DUNAGAN, P. JOSHI, et C. GUO. « A Generic Application-Level Protocol Analyzer and its Language ». Dans *14th Annual Network & Distributed System Security Symposium*, 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, 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, 2006.

[BHM+04] D. BOOTH, H. HAAS, F. MCCABE, E. NEWCOMER, M. CHAMPION, C. FERRIS, et D. ORCHARD. « Web Services Architecture ». W3C Note NOTE-ws-arch-20040211, World Wide Web Consortium, 2004.

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

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

[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.

[BRL01] E. BIAGIONI, Harper R., et P. LEE. « A Network Protocol Stack in Standard ML ». *Higher Order Symbol. Comput.*, 14(4) :309–356, 2001.

[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. IEEE, 2007.

[BSP+05] B.N. BERSHAD, S. SAVAGE, P. PARDYAK, E. GÜN SIRER, M.E. FIUCZYNSKI, D. BECKER, C. CHAMBERS, et S. EGGERS. « Extensibility, Safety and Performance in the SPIN Operating System ». Dans *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–283. ACM, 2005.

[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.

[CCCY02] V. CARDELLINI, E. CASALICCHIO, M. COLAJANNI, et P. YU. « The state of the art in locally distributed Web-server systems ». *ACM Comput. Surv.*, 34(2) :263–311, 2002.

[CE02] M. CORTES et J. R. ENSOR. « Narnia : a virtual machine for multimedia communication services ». Dans *Proceedings of the Fourth International Symposium on Multimedia Software Engineering*, pages 246 – 254, 2002.

[CEE04] M. CORTES, J.R. ENSOR, et J.O. ESTEBAN. « On SIP Performance ». Rapport Technique, Bell Labs Technical Journal 3, 2004.

[CEI+07] A. CHANDER, D. ESPINOSA, N. ISLAM, P. LEE, et G. NECULA. « Enforcing resource bounds via static verification of dynamic checks ». *ACM Trans. Program. Lang. Syst.*, 29(5) :28, 2007.

[CFSD90] J. D. CASE, M. FEDOR, M. L. SCHOFFSTALL, et C. DAVIN. « RFC 1157 : Simple Network Management Protocol (SNMP) », 1990.

[CGP99] E. CLARKE, O. GRUMBERG, et D. PELED. *Model Checking*. The MIT Press, 1999.

[Cle88] J. Graig CLEAVELAND. « Building Application Generators ». *IEEE Software*, 1988.

[CLLM04] C. CONSEL, J.L. LAWALL, et A.-F. LE MEUR. « A Tour of Tempo : A Program Specializer for the C Language ». *Science of Computer Programming*, 52 :341–370, 2004.

[CLRC05] C. CONSEL, F. LATRY, L. RÉVEILLÈRE, et P. COINTE. « A Generative Programming Approach to Developing DSL Compilers ». Dans *Fourth International Conference on Generative Programming and Component Engineering (GPCE)*, volume 3676 de *Lecture Notes in Computer Science*, pages 29–46. Springer-Verlag, 2005.

[CM98] C. CONSEL et R. MARLET. « Architecturing software using a methodology for language development ». Dans *Proceedings of the 10th International Symposium on Programming Language Implementation and Logic Programming*, volume 1490 de *Lecture Notes in Computer Science*, pages 170–194, 1998.

[CO97] D. CROCKER, ED. et P. OVERELL. « RFC 2234 : Augmented BNF for Syntax Specifications : ABNF », 1997.

[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.

[Cou] COUNTERPATH. « X-lite, a fully functionning softphone ». http ://www.counterpath.com/xlitedownload.html.

[CRH01] J. COSTA-REQUENA et T. HAITAO. « Enhancing SIP with spatial location for emergency call services ». Dans *Proceedings of the 10th IEEE International Conference on Computer Communications and Networks*, pages 326–333, 2001.

[CRL96] S. CHANDRA, B. RICHARDS, et J.R. LARUS. « Teapot : Language Support for Writing Memory Coherence Protocols ». Dans *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 237–248, 1996.

[Der] H. DERBY. « The Performance of FoxNet 2.0 ». citeseer.ist.psu.edu/derby99performance.html.

[dGSA07] G. de GEEST, A. SAVELKOUL, et A. ALIKOSKI. « Building a framework to support Domain-Specific Language evolution using Microsoft DSL Tools ». Dans *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modelling*, 2007.

[DLS78] R. A. DEMILLO, R. J. LIPTON, et F. G. SAYWARD. « Hints on Test Data Selection : Help for the Practicing Programmer ». *Computer*, 11(4) :34–41, 1978.

[DRM04] J. DERUELLE, M. RANGANATHAN, et D. MONTGOMERY. « Programmable Active Services for JAIN SIP ». Rapport Technique, National Institute of Standards and Technology, June 2004.

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

[dtdb] « XML Schema ». http ://www.w3.org/XML/Schema.

[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. ACM, 2007.

[EH07b] T. EKMAN et G. HEDIN. « The JastAdd system : modular extensible compiler construction ». *Sci. Comput. Program.*, 69(1-3) :14–26, 2007.

[Eva67] G. EVANS. « Experience gained from the American Airlines SABRE system control program ». Dans *Proceedings of the 1967 22nd national conference*, pages 77–83. ACM, 1967.

[Exm] « Exmap-console ». http ://projects.o-hand.com/exmap-console.

[exo03] « The eXtended Osip Library ». http ://savannah.nongnu.org/projects/exosip/, 2003.

[FB96] M. E. FIUCZYNSKI et B. N. BERSHAD. « An Extensible Protocol Architecture for Application-Specific Networking ». Dans *USENIX Annual Technical Conference*, pages 55–64, 1996.

[Fel90] M. FELLEISEN. « On the Expressive Power of Programming Languages ». Dans *ESOP '90 3rd European Symposium on Programming, Copenhagen, Denmark*, volume 432, pages 134–151. Springer-Verlag, 1990.

[FG05] K. FISHER et R. GRUBER. « PADS : a domain-specific language for processing ad hoc data. ». Dans *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 295–304. ACM, 2005.

[Fou] Mozilla FOUNDATION. « Firefox Web Browser ». http ://en.www.mozilla.com/en/firefox/.

[fre] « FreeRTOS ». http ://www.freertos.org.

[Fre87] P. FREEMAN. « A Conceptual Analysis of the Draco Approach to Constructing Software Systems ». *IEEE Transactions on Software Engineering*, 13(7) :830–844, 1987.

[GHL+92] R. W. GRAY, V. P. HEURING, S. P. LEVI, A. M. SLOANE, et W. M. WAITE. « Eli : A Complete, Flexible Compiler Construction System ». *Comm. of the ACM*, pages 121–131, 1992.

[GJS96] J. GOSLING, B. JOY, et G. STEELE. *The Java Language Specification*. Addison-Wesley, 1996.

[Goo] GOOGLE. « GoogleTalk, a Google approach to instant communications ». http ://www.google.com/talk/.

[Gro] Exolab GROUP. « The Castor project ». http ://www.castor.org.

[hel] « The Helix Multimedia server ». http ://www.realnetworks.com/industries/mobile/products.

[HF92] P. HUDAK et J. H. FASEL. « A gentle introduction to Haskell ». *SIGPLAN Not.*, 27(5) :1–52, 1992.

[HG07] M. HIRZEL et R. GRIMM. « Jeannie : granting java native interface developers their wishes ». Dans *OOPSLA '07 : Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 19–38. ACM, 2007.

[HHS01] J. HO, J. HU, et P STEENKISTE. « A conference gateway supporting interoperability between SIP and H.323 ». Dans *MULTIMEDIA '01 : Proceedings of the ninth ACM international conference on Multimedia*, pages 421–430. ACM, 2001.

[HL94] R. HARPER et P. LEE. « Advanced Languages for Systems Sofware – The Fox Project in 1994 ». Technical Report CMU-CS-94-104, School of Computer Science, Carnegie Mellon University, 1994.

[HLP98] R. HARPER, P. LEE, et F. PFENNING. « The Fox Project : Advanced Language Technology for Extensible Systems ». Rapport Technique CMU-CS-98-107, School of Computer Science, Carnegie Mellon University, 1998.

[Hol97] G. HOLZMANN. « The Model Checker SPIN ». *Software Engineering*, 23(5) :279–295, 1997.

[HP91] N. C. HUTCHINSON et L. L. PETERSON. « The *x*-Kernel : An Architecture for Implementing Network Protocols ». *IEEE Transactions on Software Engineering*, 17(1) :64–76, 1991.

[HP00] H. HOSOYA et B. PIERCE. « XDuce : A Typed XML Processing Language' ». Dans *Int'l Workshop on the Web and Databases (WebDB)*, 2000.

[Hud96] P. HUDAK. « Building Domain-Specific Embedded Languages ». Dans *ACM Workshop on Software Engineering and Programming Languages*, 1996.

[ich] « Ichat ». http ://www.apple.com/macosx/features/ichat.html.

[Jav] « JavaCC ». https ://javacc.dev.java.net/.

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

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

[Kam96] S. KAMIN. « Standard ML as a meta-programming language ». Rapport Technique, University of Illinois, Urbana-Champaign, 1996.

[KD88] B KERNIGHAN et M. R. DENNIS. *The C Programming Language*. Prentice Hall Professional Technical Reference, 1988.

[Kic96] G. KICZALES. « Aspect-Oriented Programming ». http ://www.parc.xerox.com/spl/-projects/aop/, 1996.

[KMB+96] R. KIEBURTZ, L. MCKINNEY, J. BELL, J. HOOK, A. KOTOV, J. LEWIS, D. OLIVA, T. SHEARD, I. SMITH, et L. WALTON. « A software engineering experiment in software component generation ». Dans *Proceedings of the 18th IEEE International Conference on Software Engineering ICSE-18*, pages 542–553, 1996.

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

[Kri06] S. KRISHNAMURTHY. « TinySIP : Providing Seamless Access to Sensor-based Services ». Dans *3rd International Conference on Mobile and Ubiquitous Systems : Networking and Services*, numéro 4611 dans Lecture Notes in Computer Science, pages 1–9, 2006.

[KT03] A. KRIEGEL et B TRUKHNOV. *SQL Bible*. Wiley, 2003.

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

[Lin] « Linphone, an open-source SIP video-phone for Linux and Windows ». http ://www.linphone.org/.

[liv] « LiveMedia : Streaming Media ». http ://www.livemediacast.net/about/library.cfm.

[LMHR05] S. LEGGIO, J. MANNER, A. HULKKONEN, et K. RAATIKAINEN. « Session Initiation Protocol deployment in ad-hoc networks : a decentralized approach ». Dans *2nd International Workshop on Wireless Ad-hoc Networks*, 2005.

[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.

[Mad07] A. MADHAVAPEDDY. « *Creating High-Performance, Statically Type-Safe Network Applications* ». PhD thesis, Cambridge University, 2007.

[MC00] P. J. MCCANN et S. CHANDRA. « PacketTypes : Abstract specifications of network protocol messages ». Dans *ACM SIGCOMM 2000 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 321–333, 2000.

[McC85] R. MCCAIN. « Reusable Software Component Construction : A Product-Oriented Paradigm ». Dans *Proceedings of the 5th AiAA/ACM/NASA/IEEE Computers in Aerospace Conference*, 1985.

[MCM+00] G. MULLER, C. CONSEL, R. MARLET, L.P. BARRETO, F. MÉRILLON, et L. RÉVEILLÈRE. « Towards Robust OSes for Appliances : A New Approach Based on Domain-Specific Languages ». Dans *Proceedings of the ACM SIGOPS European Workshop 2000 (EW2000)*, 2000.

[ME04] M. MUSUVATHI et D. ENGLER. « Model checking large network protocol implementations ». Dans *NSDI'04 : Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 12–12. USENIX Association, 2004.

[MFW+07] Y. MANDELBAUM, K. FISHER, D. WALKER, M. FERNANDEZ, et A. GLEYZER. « PADS/ML : a functional data description language ». *SIGPLAN Not.*, 42(1) :77–83, 2007.

[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.

[MHD+07] A. MADHAVAPEDDY, A. HO, T. DEEGAN, D. SCOTT, et D. SOHAN. « Melange : Creating a "Functional" Internet ». Dans *Proceedings of the EuroSys 2007, 2nd. EuroSys Conference*, pages 101–114, 2007.

[Mica] MICROSOFT. « Internet Explorer ». http ://www.microsoft.com/windows/products/default.mspx.

[Micb] « Microsoft Invisible Computing ». http ://research.microsoft.com/invisible/.

[MLBS03] G. MULLER, J. L. LAWALL, L. P. BARRETO, et J.-F. SUSINI. « A Framework for Simplifying the Development of Kernel Schedulers : Design and Performance Evaluation ». Technical report 03/2/INFO, Ecole des Mines de Nantes, 2003.

[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, 2001.

[Mos04] P. MOSSES. « Modular Language Descriptions ». Dans *GPCE'04 : Third Internation Conference on Generative Programming and Component Engineering*, pages 489–490, 2004.

[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, 2000.

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

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

[Pax99] V. PAXSON. « Bro : a system for detecting network intruders in real-time ». *Computer Networks*, 31(23–24) :2435–2463, 1999.

[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, IPTC omm*, pages 79–88. ACM Press, 2007.

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

[PDC92] T. J. PARR, H. G. DIETZ, et W. E. COHEN. « PCCTS reference manual : version 1.00 ». *SIGPLAN Not.*, 27(2) :88–165, 1992.

[POJK03] A. PELINESCU-ONCIUL, J. JANAK, et Jiri KUTHAN. « SIP Express Router (SER) ». *IEEE Network Magazine*, 17(4) :9, 2003.

[Pos80] J. POSTEL. « RFC 768 : User Datagram Protocol », 1980.

[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.

[PPSP06] R. PANG, V. PAXSON, R. SOMMER, et L. PETERSON. « binpac : a yacc for writing application protocol parsers ». Dans *Proceedings of the 6th ACM SIGCOMM on Internet measurement*, pages 289–300, 2006.

[PQ95] T. PARR et R. QUONG. « ANTLR : A predicated LL (k) parser generator », 1995.

[QPP02] X. QIE, R. PANG, et L. PETERSON. « Defensive Programming : Using an Annotation Toolkit to Build DoS-Resistant Software ». Dans *5th Symposium on Operating System Design and Implementation (OSDI 2002)*, pages 45–60, 2002.

[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. IEEE Computer Society, 2001.

[Ros02] ROSENBERG, J. ET AL.. « SIP : Session Initiation Protocol ». RFC 3261, IETF, 2002.

[SA04] P. SAINT-ANDRE. « RFC 3920 : Extensible Messaging and Presence Protocol (XMPP) : Core », 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.

[SBRA07] P. STUEDI, M. BIHR, A. REMUND, et G. ALONSO. « SIPHoc : Efficient SIP Middleware for Ad Hoc Networks ». Dans *Proceedings of the 8th ACM/IFIP/USENIX International Conference on Middleware*, 2007.

[Sch86] D.A. SCHMIDT. *Denotational Semantics : a Methodology for Language Development*. Allyn and Bacon, Inc., 1986.

[Sch97] F. SCHRO. « The GENTLE Compiler Construction System », 1997.

[sea] « Seagull, traffic generator ». http ://gull.sourceforge.net/.

[SHJ+06] R. SPARKS, A. HAWRYLYSHEN, A. JOHNSTON, J. ROSENBERG, et H. SCHULZRINNE. « Session Initiation Protocol (SIP) Torture Test Messages ». Internet Engineering Task Force : RFC 4475, 2006.

[sof06] « Sofia-SIP ». http ://opensource.nokia.com/projects/sofia-sip/, 2006.

[Spe] A. SPENCER. « Asterisk : The Open Source PBX ». http ://www.asterisk.org.

[SSP+96] E. SIRER, S. SAVAGE, P. PARDYAK, G. DEFOUW, et Bershad B.. « Writing an Operating System Using Modula-3 ». Dans *Workshop on Compiler Support for Systems Software*, 1996.

[sto] « Streaming Text Orientated Messaging Protocol site ». http ://stomp.codehaus.org/Protocol.

[TBS98] W. TAHA, W. BENAISSA, et T. SHEARD. « Multi-Stage Programming : Axiomatization and Type Safety ». Dans *Automata, Languages and Programming, 25th International Colloquium (ICALP'98)*, volume 1443 de *Lecture Notes in Computer Science*, pages 918–929, 1998.

[TCM98] S. THIBAULT, C. CONSEL, et G. MULLER. « Safe and Efficient Active Network Programming ». Dans *17th IEEE Symposium on Reliable Distributed Systems*, pages 135–143, 1998.

[Tea] The Coq Development TEAM. « The Coq Proof Assistant ». http ://coq.inria.fr/.

[Thi98] S. THIBAULT. « *Domain-Specific Languages : Conception, Implementation and Application* ». PhD thesis, University of Rennes 1, France, 1998.

[Tho99] J.P. THOMESSE. « Histoire, état de l'art et perspectives des réseaux de terrain ». Dans *Proceedings of European Symposium INNOCAP'99*, 1999.

[Tur37] A. M. TURING. « On computable numbers, with an application to the *Entscheidungsproblem* ». *Proceeding of the London Mathematical Society, series 2*, 42 :230–265, 1936–1937.

[UK94] B. UPENDER et P KOOPMAN. « Communications Protocols for Embedded Systems ». *ACM Transactions on Programming Languages and Systems*, 11(7) :46–58, 1994.

[Var05] G. VARGHESE. *Network Algorithmics, an interdisciplinary approach to designing fast networked devices*. Morgan Kaufmann Publishers, 2005.

[vdBHKO02] M. van den BRAND, J. HEERING, P. KLINT, et P. A. OLIVIER. « Compiling language definitions : the ASF+SDF compiler ». *ACM Trans. Program. Lang. Syst.*, 24(4) :334–368, 2002.

[vDK97] A. van DEURSEN et P. KLINT. « Little languages : little maintenance ? ». Dans *1st ACM-SIGPLAN Workshop on Domain-Specific Languages*. Computer Science Technical Report, University of Illinois at Urbana-Champaign, 1997.

[vlc] « The VideoLAN Project ». http ://www.videolan.org/vlc/.

[voi] « VoIP protocols ». http ://www.protocols.com/pbook/VoIP.htm.

[VWBH06] E. VAN WYK, D. BODIN, et P. HUNTINGTON. « Adding Syntax and Static Analysis to Libraries via Extensible Compilers and Language Extensions ». Dans *Proc. of LCSD 2006, Library-Centric Software Design*, 2006.

[VWKSB07] E. VAN WYK, L. KRISHNAN, A. SCHWERDFEGER, et D. BODIN. « Attribute Grammar-based Language Extensions for Java ». Dans *European Conference on Object Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer Verlag, 2007.

[VWS07] E. VAN WYK et A. SCHWERDFEGER. « Context-Aware Scanning for Parsing Extensible Languages ». Dans *Intl. Conf. on Generative Programming and Component Engineering, GPCE 2007*. ACM Press, 2007.

[Wad90] P. WADLER. « Comprehending monads ». Dans *LFP '90 : Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78. ACM, 1990.

[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. WEISS. « Commonality Analysis : A Systematic Process for Defining Families ». *Lecture Notes in Computer Science*, 1429, 1998.

[Wil04] D. WILE. « Lessons Learned from Real DSL Experiments ». *Sci. Comput. Program.*, 51(3) :265–290, 2004.

[WSKW07] S. WANKE, M. SCHARF, S. KIESEL, et S. WAHL. « Measurement of the SIP Parsing Performance in the SIP Express Router ». Dans *Dependable and Adaptable Networks and Services*, numéro 4606 dans Lecture Notes in Computer Science, pages 103–110, 2007.

Quatrième partie

Annexes

Annexe A

The Zebu Language

Notation

The syntax of the Zebu language is given in a BNF-like notation. Terminal symbols are set in a `typewriter font`. Non-terminals are set in an *italic font*. The vertical bar | denotes an alternative in a rule. Parentheses (...) denote grouping. Parentheses with a trailing star sign $(\ldots)^*$ denote zero, one or several occurrences of the enclosed item. Parentheses with a trailing plus sign $(\ldots)^+$ denote one or several occurrences of the enclosed item. Parentheses with a trailing question mark sign $(\ldots)^?$ denote an optional item.

A.1 Introduction

Zebu is a domain-specific language for specifying the protocol-handling layer of network applications that use textual application protocols *à la* HTTP. The Zebu syntax is very close to that of ABNF, a high level formalism used to specify protocols. By annotating the original ABNF specification of a protocol, the Zebu user can dedicate the protocol-handling layer to the needs of a given application.

The Zebu compiler first checks the annotated specification for inconsistencies and then generates a protocol-handling layer according to the provided annotations. This protocol-handling layer is made of a set of data structures that represent a message, a parser that fills in these data structures that represent a message and various stub functions to access these data structures or drive the parsing of a message.

A.2 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 otherwse confused as a single identifier, literal or keyword. Apart from that, they are ignored.

Comments

Comments are ABNF-like comments. All characters from the characters `;` until the end of the line are considered as part of a comment. Comments are treated as blanks.

Identifiers

Identifiers are a sequence of letters, digits and _ (the underscore character) starting with a letter. 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 and identifier.

Keywords

The identifiers below are reserved keywords :

```
message     request     response    requestLine statusLine
header      struct      enum        union       uint16
uint32      mandatory   ReadOnly    multiple
```

The following character sequences are also reserved :

```
{ }   [ ]   ( )
:     =       ;
```

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.3 Protocol-Handling Layer Specification

A Zebu protocol-handling layer specification is based on ABNF, as found in RFCs, and extends it with annotations indicating which message fields should be stored in data structures and other semantic attributes. These annotations express both constraints derived from the protocol RFC and constraints that are specific to the the target application.

Figure A.4 describes the BNF syntax of a device specification. A protocol-handling layer specification is introduced by the `message` keyword and consists of an identifier followed by one or more Zebu rule definition. The identifier is the name of the protocol considered.

```
spec           ::= message ident { message-body+ }
message-body   ::= request-block
                 | response-block
                 | zabnf
                 | header
                 | constraint
request-block  ::= request { request-body+}
request-body   ::= request-line
                 | header
                 | zabnf
                 | {constraint}
request-line   ::= requestLine = rule-defBody {attribute}*
response-block ::= response {response-body+}
response-body  ::= status-line
                 | header
                 | zabnf
                 | {constraint}
status-line    ::= statusLine = rule-defBody {attributes}*
zabnf          ::= layout-attribute? ident = rule-defBody
                 | ident =/ rule-defBody
header         ::= header ident ({ rule-defBody })? = rule-defBody attributes*
rule-defBody   ::= element (/ elements)*
elements       ::= element element+
                 | element (/ elements)*
element        ::= repeat? element-def layout-member?
repeat         ::= digit
                 | digit? * digit?
element-def    ::= ident
                 | char-val | num-val | bin-val | dec-val | hex-val | prose-val
                 | ( rule-defBody )
                 | [ rule-defBody ]
layout-member  ::=  : ident as type-attr?
type-attr      ::= uint8
                 | uint16
                 | uint32
```

FIG. A.1 – Zebu BNF Syntax

A.3.1 Request/Response Blocks

The request (response) block of a Zebu specification surrounds rules and constraints specific to requests (resp. responses). The `requestLine` rule represents the first line of a request, so does the `statusLine` rule for a response. Headers rules that appear in such a block represent headers that could only appear either in requests or responses.

A.3.2 Regular Rule Definition

Regular rule definitions are equivalent to classic ABNF rule definition. A rule definition consists of a rule name followed by a = (the equal character), then an alternation of successive elements defines the rule body. An element that refers to another rule is called a *non-terminal* as opposed to a *terminal*.

A.3.3 Headers Definition

Header definition rules are *special* rules. Hence, the `header` keyword precedes each header declaration. The rule `ident` is used as the name of the header, lines whose beginning match the rule name would be parsed as this particular kind of header. Alternative identifiers provided between braces can also be used.

A.3.4 Layout Member

Each grammar element can be annotated by a `:` (the colon character) followed by an identifier to be made available in the data structure representing a message. Optionally, a type attribute can be added both to cast the parsed element to another type than string and to set value constraints.

A.3.5 Attributes

Attributes can be added to grammar rules as constraints or optimisability opportunities. Currently, four attributes are supported. `ReadOnly`, `mandatory` and `multiple` can only be applied to header rules. `ReadOnly` prevents modifications of an headers which is supposed to be constant throughout the processing. `mandatory` implies that the annotated header must be present in every message. Finally, `multiple` specifies that such an header can appear several times in one message. The `lazy` attribute is different since it enables to perform an incremental parsing for some rules. These rules are going to be parsed only when requested by the application.

attributes	::=	*attribute* (`;` *attributes*)$^{?}$
	|	*attribute*
attribute	::=	`ReadOnly` | `mandatory` | `multiple` | `lazy`
layout-attribute	::=	`struct`
	|	`union`
	|	`enum`
	|	`uint8`
	|	`uint16`
	|	`uint32`

FIG. A.2 – Attributes Specification

A.4 Constraints

A Zebu programmer can add various structural constraints, which are usually specified in natural language alongside the ABNF protocol specification. Two kinds of constraints exist : value constraints and presence constraints. An element value can depend on the value of another element's value.

Constraints can express presence depedencies, an header can become mandatory according to the presence or the value of some message elements.

```
constraint          ::=  valueConstraint
                       | presenceConstraint
valueConstraint     ::=  canonId op rhs
presenceConstraint  ::=  headerName mandatory when canonId op rhs
rhs                 ::=  canonId
                       | char-val | num-val | bin-val | dec-val | hex-val | prose-val
canonId             ::=  ident(. ident)+
op                  ::=  =
                       |  !=
                       |  >
                       |  <
```

FIG. A.3 – Constraints Specification