Building Efficient Distributed Systems: A Domain-Specific Language Based Approach

Laurent Réveillère

Université de Bordeaux 1
Laboratoire Bordelais de Recherche en Informatique

HABILITATION À DIRIGER DES RECHERCHES

au titre de l'école doctorale de Mathématiques et Informatique de Bordeaux

présentée par
Laurent RÉVEILLÈRE

Titre:

Building Efficient Distributed Systems: A Domain-Specific Language Based Approach

Développement de systèmes distribués efficaces: une approche fondée sur les langages métiers

soutenue le 23 Novembre devant la commission d'examen

Pr.	Yolande	BERBERS	Rapporteurs
Pr.	Laurence	DUCHIEN	
Pr.	Jan	VITEK	
Pr.	Xavier	BLANC	Examinateurs
Dr.	Julia	LAWALL	
Dr.	Christine	MORIN	
Dr.	Gilles	MULLER	

Summary

In recent years, many distributed systems have evolved to cope with the convergence of their domain and computer networks. As an example, Internet telephony has drastically change the face of the telecommunications domain in the last decade by introducing many new services based on Web services and databases. Distributed systems that provide advanced services must be efficient to be able to treat a large number of users and must be robust to face various attacks targeting the service itself or the underlying platform. However, building efficient and robust distributed systems requires an intimate knowledge of the relevant protocols and a substantial understanding of low-level system and network programming, which can be a challenge for many programmers. Nevertheless, the development process of distributed systems still remains rudimentary and requires a high level of low-level expertise.

In this thesis, we show that domain-specific languages (DSLs) can successfully reduce the level of expertise required to build efficient and robust distributed systems, making service programming in the reach of average developers. We present three contributions in this area. Our first contribution targets the creation of telephony services based on the SIP protocol. We have defined a domain-specific virtual machine for SIP and a DSL named SPL, providing the programmer with high-level notations and abstractions dedicated to telephony service development. The robustness of SPL has been a key factor in expediting service deployment. A variety of services have been written in SPL, demonstrating the usability and ease of programming of the language. Our second contribution is Zebu, a DSL-based approach for the development of network application protocol-handling layers. We have demonstrated, through various experiments, that this approach is a reliable alternative to manual development in the context of protocol-handling layers. Zebu generated code has good performance and has a significantly lower memory footprint than comparable existing manually encoded solutions, while guaranteeing the robustness and performance properties. The third contribution of this thesis is z2z, a generative approach to gateway construction that enables communication between devices that use incompatible protocols. Z2z includes a compiler that checks essential correctness properties, and a runtime system that hides low-level details from the gateway programmer. We have used z2z to automatically generate gateways between various incompatible protocols. The generated gateways run with a low runtime memory footprint, with essentially no runtime overhead.

Contents

Chapter 1

Introduction

Since I obtained my PhD in December 2001, I have been working on four research topics: domain-specific language (DSL) engineering, DSLs for building streaming applications, DSLs for building efficient distributed systems, and protocol interoperability. I briefly describe below the main results achieved for each topic.

Domain-specific language engineering. I have contributed to the design of a new methodology to develop DSL compilers, centered around the use of generative programming tools. Our approach enables the development of a DSL compiler to be structured on facets that represent dimensions of compilation. Each facet can then be implemented in a modular way, using generative programming tools. These tools enable modeling the high-level nature of DSLs and the richness of the builtin domain-specific information in terms of program generation. The originality of our methodology is that it modularizes the program generation process of a DSL compiler. This work was published at Dagstuhl 2004 [29] and GPCE 2005 [28].

DSL for building streaming applications. I have participated in the design of Spidle, a domain-specific language for specifying streaming applications. The development of multimedia streaming applications is becoming an increasingly important software activity. More and more new formats compete to structure the main media types, creating an explosion in format converters. The need for continuous innovation in the multimedia device industry has shifted an increasing part of stream processing from hardware to software, to shorten the time-to-market. However, due to the lack of programming language support, the development of streaming applications tends to be labor-intensive, cumbersome and error-prone. The originality of our language-based approach is that it offers high-level, declarative constructs compared to general purpose languages, and it improves robustness by enabling a variety of verifications to be performed. Compiled Spidle programs have performance that is roughly comparable to compiled equivalent C programs. This work was published at GPCE 2003 [27].

DSLs for building efficient distributed systems. I have participated in the design of domain-specific languages that are used as core building blocks in efficient distributed systems. In particular, we designed SPL, a language for creating telephony services based on the SIP protocol. This work was published at ICIN 2006 [18], ICC 2006 [17], IPTCComm 2007 [78], and led to an European patent in 2008 [19]. Subsequently, to ease the development of protocol-handling layers of network applications that use textual HTTP-like application protocols, such as SIP, we have designed the Zebu language. This work was published at SRDS 2007 [21] and TSE 2011 [22]. We have also designed z2z, a generative approach to gateway construction that enables communication between devices that use incompatible protocols. The originality of our approach is that it simplifies gateway construction, a problem that has not been considered by previous frameworks for gateway development. We have shown that gateways generated from z2z are robust and efficient. This work was published at Middleware 2009 [15] and Middleware 2010 [7].

Protocol interoperability. I have participated in the design of the Starlink framework, a general framework into which runtime generated interoperability logic can be deployed to *connect* two heterogeneous protocols. The key contribution of Starlink is the ability to create and deploy runtime solutions to interoperability using only high-level models of protocol behavior. This work was published at ICDCS 2011 [12]. We have extented the Starlink framework to model both applications and protocols such that the code to interoperate can be generated dynamically. The originality of our approach lies in the fact that it does not focus on only one of the relevant dimensions (*i.e.* application or protocol) while making assumptions about the common nature of the other. This work was published at Middleware 2011 [13].

In this document, I focus on the work I carried out on DSLs for building efficient distributed systems. In the remainder of this chapter, I first present domain-specific languages and describe the context of this work. Then I briefly introduce the main contributions. Finally, I describe the outline of the rest of the document.

1.1 Domain-specific languages

A domain-specific language is a programming or specification language dedicated to a particular domain or problem. A DSL provides appropriate built-in abstractions and notations; it is often small, more declarative than imperative, and less expressive than a general-purpose language. Literally hundreds of DSLs are in existence today. Some are distributed worldwide and used on a daily basis, *e.g.*, SQL, Unix shells, makefiles, etc. DSLs have been used in various domains such as graphics [38], financial products [4], telephone switching systems [49], protocols [24, 95], operating systems [84], device drivers [71], routers in networks [95] and robot languages [8]. As a result of these successes, DSLs have recently received a lot of attention from both the research and industrial communities [30, 48]. The following points explain why DSLs are more attractive than general-purpose languages (GPLs) for a variety of applications.

Easier programming. Because of appropriate abstractions, notations and declarative formulations, a DSL program can more concise and readable than its GPL counterpart. Hence, development time is shortened and maintenance can be improved. As programming focuses on what to compute as opposed to how to compute, and thus the user does not have to be a skilled programmer. Specific optimization strategies can be implemented in the DSL compiler not only to offer performance but also to systematize it.

Systematic re-use. Most GPL programming environments include the ability to abstract common operations into libraries. However, re-use of libraries is left to the programmer. In contrast, a DSL offers guidelines and built-in functionalities which enforce re-use. Additionally, a DSL captures domain expertise, either implicitly by hiding common program patterns in the DSL implementation, or explicitly by exposing appropriate parameterization to the DSL programmer. Thus, the programmer necessarily re-uses library components and domain expertise.

Improved safety. DSLs enable more properties about programs to be automatically checked. In contrast to a GPL, the semantics of a DSL can be restricted to make decidable some properties that are critical to a domain [95]. Detecting errors early in the development process also improves productivity. In addition, as re-use is not only improved but systematized, DSL programs rely on components that are frequently used and thus well tested.

In practice, the formulation in terms of a DSL suggests an attractive way to architecture software to implement a program family. In the remainder of this document, we present contributions that target program families in the context of distributed systems.

1.2 Context

Distributed systems often provide advanced services to cope with user expectations. However, building such systems is challenging because they must efficiently process various kind of network messages, support service development without sacrifying robustness, and allow the composition of heterogeneous independently developed services.

Distributed systems rely on protocols to define the set of rules governing communication between network applications. This communication is managed by the part of the application known as the protocol-handling layer, which 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 survive various attacks targeting the application itself or the underlying platform. Nevertheless, the process of developing code for this layer still remains rudimentary and requires a high level of expertise in networking and system programming.

Many distributed systems have evolved to cope with the convergence of their domain and computer networks. As an example, Internet telephony has drastically change the face of telecommunications in the last decade. Telephony over IP enriches the domain of telecommunications with a host of new functionalities such as Web services and databases and makes service creation a key technology enabler. Service creation thus becomes a vast application area, ranging from organizing the telephone communications within a small business to managing calls in telemarketing centers. However, programming these kind of services requires an intimate knowledge of a variety of protocols and technologies, as well as expertise in system programming, networking, and telephony protocol APIs. These requirements make service engineering an overwhelming challenge for many programmers.

Nowadays, independently developed distributed systems often must be composed together to provide advanced services. However, existing systems are highly heterogeneous in their interaction methods making such composition challenging. Applications and systems are developed using a multitude of incompatible protocols that limit interoperability, and thus the practical benefit of systems composition. Protocol standardization should address this issue but has been demonstrably ineffective in practice. Indeed, new competing protocols are frequently introduced to address the needs of new application domains (e.g. sensors, ad-hoc networks, Grid Computing, Cloud Computing, etc.), whereas standardization is slow in comparison. To address this issue, gateways are needed to translate between the various kinds of heterogeneous protocols. Gateway construction, however, requires an intimate knowledge of the relevant protocols and a substantial understanding of low-level network programming, which can be a challenge for many application programmers.

1.3 Contributions

This thesis presents three contributions to building efficient distributed systems. I briefly introduce them below, before presenting them in detail in the next chapters.

The SPL Language for Internet-Telephony Service Creation The first contribution presented in this thesis is SPL, an expressive language to ease the development of telephony services without sacrificing safety. SPL relies on a domain-specific virtual machine for SIP (SIP VM) that provides a design framework for service development based around the notion of a session. The design of SPL has resulted from a thorough analysis of the requirements for a language for programming Internet telephony services. Because SPL offers high-level abstractions, it frees the service developer from low-level programming details and complexities of the underlying technologies. SPL guarantees critical properties that cannot be verified in general-purpose languages (GPLs), by introducing domain-specific concepts and semantic restrictions.

Although various approaches have been proposed to develop SIP-based telephony services, they are still limited. Indeed, many of these approaches are based on general-purpose programming languages

such as C, Java and C#. Platforms that rely on these languages offer the programmer a powerful but complicated access to advanced functionalities. Although expressive, these approaches cannot enforce the safe execution of a service. To overcome these limitations, some of these platforms introduce restricted scripting languages such as CPL [61] and LESS [100] for service programming. However, services written using these scripting languages are usually limited to coarse-grained processing or a selection of fixed treatments.

To address these issues, we have proposed a DSL approach for developing telephony services based on a combination of a dedicated virtual machine and a dedicated programming language. We have defined a domain-specific virtual machine, named the SIP VM, that provides a high-level interface, dedicated to service development. This virtual machine is centered around the notion of a session, consisting of a set of events and a state, which structures the development of a telephony service. Furthermore, by abstracting over the platform, the SIP VM enables services to be portable. The SIP virtual machine, combined with common programming patterns found in telephony services, forms the main ingredient in the design of SPL. SPL introduces notations and abstractions that are specific to the domain of telephony services, facilitating the development process and offering expressivity. Indeed, SPL enables a programmer to concentrate on defining *what* the service logic should do as opposed to dealing with every detail of *how* to implement it. SPL introduces notations and abstractions that are specific to the domain of SIP telephony services, facilitating the development process and offering expressiveness. The static and dynamic semantics of SPL have been formally specified, enabling a precise definition of SPL's interaction with the SIP VM.

A variety of services have been written in SPL for our university department. In these experiments, we have demonstrated the usability and ease of programming of SPL. Its robustness has been a key factor in expediting service deployment.

This work was initiated in collaboration with Charles Consel. In this context, I participated with him in the PhD supervision of Laurent Burgy [20], Fabien Latry [59], and Nicolas Palix [77] who were involved in the SPL project.

The Zebu Language for Network Protocol Message Processing The second contribution presented in this thesis is Zebu, a language to ease the development of protocol-handling layers of network applications. Zebu has been designed for HTTP-like protocols, which are text-based and line-oriented. Such protocols include SIP, RTSP, and SMTP. HTTP-like protocols are widely used, to provide interoperability between diverse applications and devices, and because the text representation makes debugging easy.

Although protocol-handling layers are a critical component of a network application, both in terms of robustness and performance, their development process remains rudimentary. Indeed, it requires a high level of expertise in multiple domains to be able to meet these robustness and performance requirements. In addition, protocols are also often extensible, as illustrated by the SIP protocol which currently has more than 150 proposed extensions. Thus, it is often not sufficient to simply develop one protocol-handling layer for a given protocol and reuse it in all applications. Instead, protocol-handling layers must be continually developed and extended to take into account new protocol variants.

To address these issues, we have proposed Zebu, a DSL to ease the development of protocol-handling layers. Zebu can generate a parser directly from an RFC specification expressed as ABNF with only a few modifications. Auxiliary constraints on message values can be added as annotations within the ABNF specification provided to Zebu. Because Zebu parsers are based on the protocol specification, they can also validate the message structure, thus improving robustness. In experiments involving messages known to be difficult to parse, we have shown that protocol-handling layers based on Zebu detect 100% of erroneous received messages, as opposed to 25% for existing, widely used, manually developed parsers. Zebu incorporates a number of state-of-the-art strategies for efficiently parsing and crafting messages for HTTP-like protocols, thus relieving the developer of the need to identify and hand-code these strategies. These optimizations address not only run-time performance but also memory usage. Experiments have shown that Zebu-generated protocol code and hand-written code with equivalent levels of robustness have the same performance.

I have initiated this work and supervised the PhD thesis of Laurent Burgy [20] on this project

The Z2z framework for Automatic Generation of Network Protocol Gateways The third contribution presented in this thesis is z2z, a generative approach to gateway construction that enables communication between devices that use incompatible protocols. A gateway provides interoperability between two fixed protocols, and thus must be developed separately for every pair of protocols between which interaction is needed. The diversity of protocols that are used in a networked environment implies that this is a substantial development task. To reduce the development effort, indirect bridges translate messages to and from a single fixed intermediary protocol. This approach, however, may limit expressiveness, as some aspects of the relevant protocols may not be compatible with the chosen intermediary protocol.

A gateway must take into account the different degrees of expressiveness of the source and target protocols and the range of communication methods that they use. Each of these issues requires substantial expertise in network programming, and the need to address both of them at once makes gateway development especially difficult.

To address these issues, we have proposed z2z, a new approach to gateway development. Our approach relies on the use of a domain-specific language (DSL) for describing protocol behaviors, message structures, and the gateway logic. The DSL relies on advanced compilation strategies to hide complex issues from the gateway developer such as asynchronous message responses and the management of dynamically-allocated memory, while remaining in a low-overhead C-based framework. We have implemented a compiler that checks essential correctness properties and automatically produces an efficient gateway implementation. We have implemented a runtime system that addresses a range of protocol requirements, such as unicast or multicast transmission, association of responses to previous requests, and management of sessions.

We show the applicability of z2z by using it to automatically generate gateways between SIP and RTSP, between SLP and UPnP, and between SMTP and SMTP via HTTP. The gateway specifications are 100-400 lines of z2z code. The generated gateways are at most 150KB of compiled C code and run with a memory footprint of less than 260KB, with essentially no runtime overhead.

I have initiated this work in collaboration with David-Yérom Bromberg who had been recruited as an associate professor at the University of Bordeaux at the beginning of the project. I am currently supervising three PhD students who are involved in the z2z project.

1.4 Outline

The thesis is organized as follow. Chapter 2 presents the SPL language for Internet-telephony service creation. Chapter 3 presents the Zebu language for network protocol message processing. Chapter 4 presents the z2z framework for networks protocol gateway generation. Finally, Chapter 5 concludes the thesis and gives some directions for future work.

Chapter 2

SPL: a Language-Based Approach for Internet-Telephony Service Creation

This chapter presents SPL (*Session Processing Language*) [18, 17, 78, 19], a domain-specific language for creating telephony services. SPL offers domain-specific constructs, abstracting over the intricacies of the underlying technologies. By design, SPL guarantees critical properties that are difficult or impossible to verify when using general-purpose languages. SPL relies on a domain-specific virtual machine, named the SIP VM. This virtual machine introduces a design framework for service development based around the notion of a session. We show that SPL and the SIP VM can be used to develop and deploy real services, demonstrating the practical benefits of our approach.

2.1 Introduction

In recent years, telephony has increasingly converged with networks and multimedia. This has led to a rapid evolution in the telephony domain. Now that telephony can interact with systems such as databases and Web services, it can offer a host of new functionalities. Meanwhile, telephony services represent a vast application area, ranging from organizing the telephone communications within a small business to telemarketing centers.

At the forefront of modern telephony is the *Session Initiation Protocol* (SIP) [89, 88]. SIP is a fast-emerging signaling protocol for Voice over IP (VoIP) and third generation mobile phones. It is standardized by the IETF[1] and adopted by the ITU.[2] This protocol enables creating, modifying and terminating a communication between parties. Communications include audio/video communications, games, and instant messaging. SIP supports user mobility by giving a user a symbolic address that can be associated with various communication devices. It also provides for user availability; that is, willingness of a callee to accept a dialog. SIP is based on a client-server model. A SIP platform is typically implemented as a distributed system that consists of servers providing the various SIP functionalities. A fast-growing number of SIP platforms is becoming available; some target telephone carriers, others small businesses. A SIP platform can either be located at the core or at the edge of the network.

A variety of approaches have been proposed for developing services targeting SIP-based systems. Many of these approaches are based on general-purpose programming languages such as C, Java and C#. Platforms that rely on these languages offer the programmer a powerful but complicated access to advanced functionalities. Although expressive, these approaches cannot enforce the safe execution of a service. To overcome these limitations, some platforms introduce a restricted scripting language such as CPL [61] and

[1] IETF: Internet Engineering Task Force.
[2] ITU: International Telecommunications Union.

LESS [100] for service programming. However, services written using these scripting languages are usually limited to coarse-grained processing or dedicated treatments. To combine expressivity and simplicity, some platforms such as the Microsoft Live Communication Server introduce hooks to shift the processing of a SIP message from a restricted scripting language (MSPL [73]) to a powerful API (C#).

Realizing the potential of SIP requires a proliferation of telephony services. Yet, the origins and combined technologies used in the telephony domain make software engineering of telephony services an overwhelming challenge. To develop a service, a programmer must have an extensive expertise in the telephony domain, SIP and related protocols, distributed programming, networking, and SIP APIs. Correct use of all of these features is paramount because telephony is often heavily relied on and a buggy service can make telephony unavailable to a specific user, or even to all the users of a platform.

This chapter

This chapter presents SPL, an expressive language to ease the development of telephony services without sacrificing safety. SPL relies on a domain-specific virtual machine for SIP (SIP VM) that provides a design framework for service development based around the notion of a session. The design of SPL has resulted from a thorough analysis of the requirements for a language for programming Internet telephony services. Because SPL offers high-level abstractions, it frees the service developer from low-level programming details and intricacies of underlying technologies. SPL guarantees critical properties that cannot be verified in general-purpose languages (GPLs), by introducing domain-specific concepts and semantic restrictions.

Contributions

The contributions of SPL are as follows.

SIP Virtual Machine. To remedy the shortcomings of existing APIs, we have defined a domain-specific virtual machine, named the SIP VM, that provides a high-level interface, dedicated to service development. This virtual machine is centered around the notion of a session, consisting of a set of events and a state, which structures the development of a telephony service. Furthermore, by abstracting over the platform, the SIP VM enables services to be portable.

Session Processing Language. The SIP virtual machine, combined with common programming patterns found in telephony services, forms the main ingredient in the design of SPL. SPL introduces notations and abstractions that are specific to the domain of telephony services, facilitating the development process and offering expressivity. The syntax of an SPL service reflects the SIP VM session structure. Each kind of session is represented by a block containing the declarations of the variables and handlers associated with the session. Indeed, SPL enables a programmer to concentrate on defining *what* the service logic should do as opposed to dealing with every detail of *how* to implement it.

Formalized semantics. SPL introduces notations and abstractions that are specific to the domain of SIP telephony services, facilitating the development process and offering expressiveness. The static and dynamic semantics of SPL have been formally specified, enabling a precise definition of SPL's interaction with the SIP VM.

Outline

The rest of this chapter is organized as follows. Section 2.2 presents SIP in more detail and defines requirements for a language dedicated to programming telephony services. Sections 2.3 and 2.4 introduce the SIP

virtual machine and SPL. Section 2.5 describes SPL safety properties. Section 2.6 describes the static and dynamic semantics of SPL and its relationship with the underlying SIP virtual machine. Section 2.7 discusses related work and presents an assessment of our approach. Finally, Section 2.8 concludes the chapter with some final remarks.

2.2 SIP Background

SIP is based on a client-server model. A SIP message can be sent or received by a client. A sent message is said to be *outgoing*; a received message is said to be *incoming*.

SIP messages SIP is a text-based protocol similar to other recent protocols such as HTTP[3] and RTSP.[4] A SIP message begins with a line indicating whether the message is a request (including a protocol method name) or a response (including a return code). A sequence of required and optional headers follows. Finally, a SIP message includes a body containing other information relevant to the message.

Logically, SIP is composed of three main entities, depicted in Figure 2.1: a *registrar server*, a *proxy server*, and a *user agent*. A registrar server allows a SIP user to record his current location. A proxy server dispatches SIP messages, whether incoming or outgoing, requests or responses. Typically, a SIP platform provides a registrar server and a proxy server in each domain to manage the messages to or from the users in a given domain. Additional proxy servers may be present in the network. Finally, a user agent exists within each communication device, and performs all SIP-related actions on its behalf.

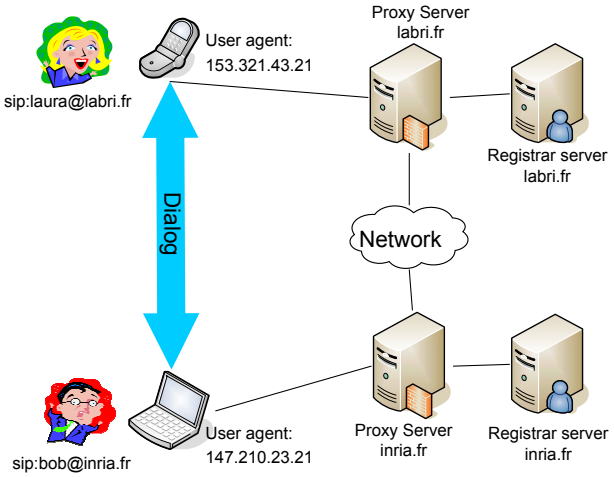


Figure 2.1: SIP architecture

To support mobility, a user is assigned a SIP URI (Uniform Resource Identifier), which is a symbolic address, analogous to an e-mail address. When a SIP proxy receives a message for a local URI, it asks the local registrar server to translate the URI into contact information for a specific user agent. A user must thus inform the registrar server of the user agent at which he would like to receive messages. As an example, the top of Figure 2.1 shows that Laura, in the domain `labri.fr`, has SIP URI `sip:laura@labri.fr`. When Laura wants to be reachable via her IP phone, she sends a `REGISTER` request including the phone's address to the `labri.fr` registrar server. If the response is the success code `OK`, subsequent calls to Laura will be routed to her IP phone. For another example, at the bottom of Figure 2.1, Bob in domain `inria.fr` has set his current location to be his laptop by communicating its IP address when registering.

[3] HTTP is the HyperText Transfer Protocol for transferring World Wide Web documents.
[4] RTSP is the Real Time Streaming Protocol for controlling the delivery of data with real-time properties.

Making a call To further present SIP, we describe the main steps involved in making a call, illustrated by Bob calling Laura once they have both registered their current location. To make the call, Bob's user agent emits an `INVITE` request including his current location and the callee's SIP URI, `sip:laura@labri.fr`. Like an e-mail message, the request is routed through a sequence of proxy servers until it reaches the domain `labri.fr`. At this point, the `labri.fr` proxy server queries the registrar server to locate Laura and forwards the request to her IP phone. Upon receipt of the request, Laura's phone starts ringing. If she picks it up, an `OK` response, including Laura's location, is sent back to the caller. Bob then sends an `ACK` message to Laura to confirm the establishment of the call. At this point a conversation can start. When either Bob or Laura wants to end the conversation, they hang up the phone, causing their user agent to send a `BYE` request to the other party.

Management of SIP communications The SIP protocol introduces four concepts used in the management of communications: transaction, dialog, subscription and registration.

- A *transaction* consists of a request and the response it triggers. For example, a `REGISTER` request sent to a user agent and its response form a transaction. A SIP message includes headers identifying the transaction it belongs to.
- A *dialog* is a SIP relationship between two user agents that persists for some time. A dialog is initiated by an `INVITE` request and confirmed by an `ACK` request. A SIP message includes a header identifying the dialog it belongs to. A dialog persists until one party sends a `BYE` request.
- A *subscription* allows a SIP user to request notification from a remote user agent when a certain event occurs. For example, one may subscribe to a presence event, to be informed when a particular user becomes available. A user interested in an event sends a `SUBSCRIBE` request to the responsible remote user agent, named the notifier. When the event occurs, the notifier sends a `NOTIFY` request to the subscriber. A subscription must be refreshed periodically.
- A *registration* comprises the dialogs created by a user via the information provided to a particular registrar server. A registration persists until the user unregisters or his registration expires.

2.2.1 SIP Services

Most SIP platforms allow users to install services on an additional SIP entity known as a *feature server*. A service processes incoming and outgoing messages, essentially acting as a router.

We illustrate the notion of SIP service with our example. Suppose that a service has been installed for Laura that allows only customer calls to reach her IP phone during office hours. When Laura's proxy server receives the `INVITE` request from Bob, it invokes the associated feature server to obtain her service. The service is launched by the SIP platform with the `INVITE` request as input. If Bob's SIP URI is not part of Laura's customer list, the service rejects it by returning an error response (*e.g.,* `UNAUTHORIZED`). If Bob is a customer, the service gives control back to the SIP platform to *forward* the `INVITE` request to Laura's user agent (*i.e.,* her IP phone). Forwarding this request creates a new transaction that ends when the response is received or confirmed. If Laura is already on the phone, her user agent sends a `BUSY HERE` response back to the service. The service may then redirect the call to her voice mail, creating another transaction.

A SIP service may require that some state be associated with a given transaction, dialog, or registration. For example, Laura's service uses a customer list, which needs to be maintained while she is registered. The use of such state requires that the proxy server retain enough information to be able to associate relevant SIP messages to an existing transaction, dialog, or registration. The treatment of such messages is said to be *stateful*. Statefulness has some cost, and consequently very frequently executed services, such as those associated with an entire domain, are typically designed without any transaction, dialog, or registration state.

2.2.2 Requirements

Requirements for telephony service programming have been already partially identified by Lennox [62]. Based on these results, we have conducted a study of different existing SIP platforms and the paradigm they introduce for developing telephony services [16]. This analysis is based on our experience on designing and implementing domain-specific programming languages (DSLs). As a result, we have revisited Lennox's requirements for a language dedicated to telephony service creation.

Accessible To enable users with limited programming experience and little domain expertise to develop their own services, a language for service creation needs to provide high-level abstractions that protect the programmer from having to be aware of the intricacies of both the platform and the underlying protocols. In doing so, an entire class of errors can be eliminated. The resulting services are easier to read, develop and maintain.

Robust and safe The telephony domain imposes stringent safety and robustness requirements. These intervene at two levels: First a telephony service should execute without runtime errors under all circumstances, to ensure continuity of service for other calls. Second, a telephony service should respect the invariants of the underlying protocol. Therefore, a language for service creation needs to fulfill these requirements, either by construction or by enabling domain-specific program analyzes.

Expressive Ensuring the safe execution of a service written in a given programming language must not be done at the expense of the expressivity and usability of that language. Service developers should be able to use mainstream general-purpose language features as well as domain-specific language constructs.

2.3 SIP Virtual Machine

All existing SIP APIs pursue the same goal: offering both in-depth and in-breadth access to the SIP protocol. In practice, this goal has translated into very large and complex APIs. Because these APIs are based on general-purpose languages, the service programmer must write repetitive glue code as the prologue and epilogue of API invocations. APIs also offer little support for structuring services or for managing service data. Finally, SIP APIs are platform-specific and thus do not permit portability.

To alleviate these problems, we define a domain-specific virtual machine for SIP, named the SIP VM, providing the programmer with a high-level portable interface dedicated to telephony service development. The SIP VM raises the level of abstraction of the SIP protocol to match the needs of telephony service developers. This VM sits within a SIP application server, between the SIP platform and the SIP services. The SIP VM mainly introduces the notions of operations, events, and sessions. We examine each of these components.

2.3.1 Operations

A SIP service typically performs some computation, forwards a SIP request and returns a SIP response. Within these computations, the SIP VM distinguishes signaling operations, which send a SIP message over the network, from non-signaling operations, which perform arithmetic and other local calculations.

Signaling operations The SIP VM defines the operation `forward` to allow a service to forward a request. To use this operation, the service provides not only a pointer to the request, but also the service's current code pointer and state. When the corresponding response is received, the SIP VM restores the code

pointer and state of the service, thus causing the execution of the service to continue from the point of forwarding the request.

After completing the processing of a request, a service has to return a SIP response. Typically, a service returns the response it received after performing a `forward` operation. However, some services must return a constant SIP response without forwarding any request, and thus must be able to forge responses. For example, a service that implements a black list must return a reject response code (*e.g.*, `486`) to any black listed caller. The SIP VM thus introduces abstractions and operations to manipulate SIP responses, including operations to forge a SIP response from the current request.

Non-signaling operations The SIP VM also defines non-signaling operations such as arithmetic and boolean operations. These operations allow services to perform various computations. In addition, the SIP VM includes an extension framework to enable a service to call external procedures. Such procedures may be available on the local machine or may be invoked via remote procedure calls.

2.3.2 Events

The SIP VM defines three kinds of events to which a service can react: *verbatim* events, *refined* events and *platform* events. When a SIP request is received, the VM generates the corresponding event. If the meaning of the SIP request is unambiguous (*e.g.*, `ACK`), then the request is transmitted as a verbatim event, *i.e.*, as is. Some requests are, however, context sensitive, and require interpretation. For example, the SIP request `INVITE` either initiates a dialog or, if used in the context of an existing dialog, modifies dialog characteristics. The VM thus refines a SIP `INVITE` request as either the event `INVITE`, in the former case, or the event `REINVITE` in the latter. A platform event notifies a service of events internal to the platform that are relevant to the service logic. For example, the platform event `unregister` indicates the expiration of a SIP user registration.

Table 2.1 lists the events provided by the SIP VM. The names of verbatim and refined events are noted in uppercase to indicate that they correspond to a SIP message. The corresponding handler code in a service must perform at least one signaling operation. The names of platform events are noted in lowercase to indicate that they do not correspond to a SIP message.

Concepts	**Events**		
	Initial	**Medial**	**Final**
Service	`deploy`		`undeploy`
Registration	`REGISTER`	`REREGISTER`	`unregister`

Dialog	`INVITE`	`CANCEL`
		`ACK`
		`REINVITE`
			`BYE`
			`uninvite`

Subscription	`SUBSCRIBE`	`RESUBSCRIBE` `NOTIFY`	`unsubscribe`

Table 2.1: Classification of SIP VM events

2.3.3 Sessions

The SIP protocol defines a session as a multimedia communication dialog. SIP requests and time-outs enable to create, confirm, modify and terminate a dialog. Therefore, they define a complete lifecycle for a dialog session.

In the context of the SIP VM, we generalize this notion of a session lifecycle to the context of a subscription, a registration, and a service. A session is thus created when subscribing to an event, registering a user, or deploying a service. It persists until un-subscribing to the event, un-registering the user or undeploying the service. The events generated by the SIP VM correspond to specific points in the lifecycle

of the various SIP concepts. The events are thus classified as initial (creation), medial (confirmation and modification) and final (termination), as shown in Table 2.1.

A service typically performs some computations before triggering a signaling action. For example, a load-balancing service for a company would compute the total time each employee has spent answering the telephone to decide who should take the next call. To allow services to manipulate session states throughout their lifecycle, the SIP VM attaches a state to a session. This state is saved and restored by the SIP VM across event notifications.

2.4 The Session Processing Language

Building on our experience in developing DSL, we propose a new language, SPL (Session Processing Language), for developing telephony services. This language includes domain-specific constructs and semantics. It is designed around the abstractions furnished by the SIP VM. This section describes the salient features of SPL. The complete grammar of SPL is provided in Appendix A.

2.4.1 Sessions

The structure of an SPL service reflects the SIP VM session structure. Each kind of session is represented by a block containing the declarations of the variables and handlers associated with the session. Sessions are organized into a hierarchy, with a service session at the root, the registration sessions created within the service session as its children, and dialog and subscription sessions at the leaves. A session at any level has access to all of the variables of its ancestor sessions.

As an example, Figure 2.2 shows the SPL service `sec_calls`, that implements a counter service. This service maintains a counter of the calls that have been forwarded to a secretary, whenever the SIP user associated with the service is unable to take the call. The counter is set to `0` when the user registers, incremented when a call is forwarded to the secretary, and logged when the user unregisters. An outermost processing block declares service variables and functions, such as the external function `log`, followed if needed by handler definitions for `deploy` and `undeploy` (absent in our example). A registration block is defined inside the processing block. In our example, the registration block defines the `cnt` variable, a `REGISTER` handler, which initializes the counter, and an `unregister` handler, which logs the counter. Finally, a dialog block is defined inside the registration block. The dialog block only declares an `INVITE` handler. When an incoming call is rejected by the user, this handler increments the `cnt` variable, which is defined in the ancestor session (*i.e.,* the registration block).

As illustrated by the counter service, SPL does not require explicit state manipulation. Specifically, the `cnt` variable is defined and used as in any programming language. Such a variable results in the creation of a state object; variable manipulations are automatically mapped by SPL into access, save and restore operations.

2.4.2 Handlers

For each event generated by the SIP VM, an SPL service may define a handler to describe how the service should react to the event. In the case of verbatim and refined events, the handler processes a complete SIP transaction from the request to the final response. Platform events do not correspond to a SIP request. Therefore, the handler for such an event only performs non-signaling operations. A handler may be designated as either `incoming` or `outgoing` to provide a specific behavior for messages received or emitted by the user associated with the service, respectively. By default, a handler treats both incoming and outgoing requests.

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

Figure 2.2: The counter service in SPL

The `INVITE` handler in Figure 2.2 (lines 17-24) illustrates SPL transaction processing for incoming `INVITE` requests. In this example, the `forward` operation (line 18) is used initially to forward an incoming call to the original recipient. This operation yields control to the SIP VM, which resumes the handler when a response is received. This response is then stored in the variable `r` and checked in line 19. SPL offers domain-specific abstractions and operations for such processing of SIP responses: Comparing a response to `/SUCCESS` checks whether the response code of the response is in the range `2xx`, while comparing a response to `/ERROR` checks whether the response code of the response is greater than `300`. In the example, if the call was not accepted, the original request is redirected to the secretary (line 21) and the new response is returned to the caller. If the call was accepted, the success response that was stored in `r` is returned directly (line 23).

As a component of a SIP application server, the SIP VM may receive requests at any time. To simplify SPL programming, the SIP VM never invokes a handler during the execution of another handler. Instead, events are queued by the SIP VM until the handler being executed yields control to the SIP VM, either due to a `forward` or a `return` operation. At this point, the SIP VM sends the request or response, respectively, out on the network and treats other pending requests. In the case of a `forward`, when a response arrives, the SIP VM resumes the handler containing the `forward` operation. The execution of a handler is thus atomic between two yield points, *i.e.*, two signaling operations.

2.4.3 Intra-Handler Control Flow

Handlers for verbatim or refined control methods typically perform some computation and then forward the SIP request. This forwarding yields control to the SIP platform, which sends the request. In existing SIP APIs, when the response is received, the service code must explicitly correlate the response with the request. Then, some computation must be performed to restore the control flow suspended at the forward point before processing the response.

One of the goals of SPL is to factorize these tedious and error-prone computations out of the service. In SPL, a handler is written as a single unit that processes a transaction from the request to the response. When a handler needs to forward a message, it uses the `forward` expression, that gives the SIP VM the current code pointer and state. When the corresponding response is received, the SIP VM restores the code

```
1  service waiting_queue {
2      [...]
3      registration {
4          [...]
5          dialog {
6              [...]
7              response incoming INVITE() {
8                  response resp = forward;
9                  if (TO == 'sip:secretary@enseirb.fr') {
10                     [...]
11                     // Session tagged as "secretary"
12                     return resp branch secretary;
13                 } else {
14                     [...]
15                     // Session tagged as "private"
16                     return resp branch private;
17                 }
18                 [...]
19             }
20
21             [...]
22
23             response incoming ACK() {
24                 branch secretary {
25                     // Specific treatment for secretary session
26                     [...]
27                 }
28                 branch private { ... }
29                 [...]
30             }
31
32             [...]
33         }
34     }
35 }
```

Figure 2.3: Inter-handler control flow

pointer and state of the service, and execution of the handler continues.

Notice that `forward` can be invoked with or without a URI. When no URI, is provided, the current request is forwarded to the the original destination (*i.e.,* Request-URI as defined in the SIP RFC 3261 [89]).

The `INVITE` handler in Figure 2.2 illustrates SPL transaction processing. In this example, an incoming call is forwarded to the user and his response is assigned to the variable `r`. The response is then checked. If the call was not accepted, the original request is redirected to the secretary and the new response is returned to the caller. If the call was accepted, the success response is returned directly.

2.4.4 Inter-Handler Control Flow

Not only is a session a design thread, but it also represents a thread of control. The SIP protocol describes a coarse-grained session control flow, *i.e.*, in a dialog control may flow from `INVITE`, to `ACK`, to `BYE`. To enhance expressiveness, SPL allows the programmer to refine the control-flow specification via a *branch* mechanism that passes control information from one handler to the next. This abstraction permits, *e.g.*, classifying a session as either personal or professional, and then introducing a logical personal or professional sub-thread across the remaining handler invocations of the session. As shown in Figure 2.3, the classification is done by adding the name of the branch to the return from the current handler (*e.g.*, line 12) and the threads are implemented in each handler using the `branch` construct (*e.g.* lines 22-25).

A branch is chosen for a session when the session is created. The branch is stored in the session state and is used to select the relevant code for each subsequent handler invocation in the session. For a service or registration session, the initial branch is `default`. On returning, a handler can specify a new branch, which overwrites the current one. When a service or registration handler is invoked, the code corresponding to the current branch is chosen, or the code corresponding to the default branch, if nothing else applies. Dialog and subscription sessions are treated similarly, except that the initial branch is inherited

from the current branch of the parent registration session and branches accumulate rather than overwrite. On invoking a method, the code corresponding to the branch appearing earliest in the accumulated sequence of branches is selected.

The branch construct is not only convenient for the programmer, it also increases the amount of information that is available to static verifications of SPL programs. Specifically, as the branch mechanism is a built-in language construct, it enables inter-handler control flow information to be determined accurately, before execution time. If this mechanism were not available, the programmer could simulate the same behavior using flags stored in the session state and conditionals in each handler testing the flag values. In this case, however, the control-flow information represented by the branches would not in general be apparent to the verifications associated with SPL (see *e.g.*, Section 2.6.1), which would result in a less precise analysis.

Branches are illustrated in Figure 2.3 by fragments of a secretary service, written in SPL. When an `INVITE` request is sent to the secretary SIP URI, the `secretary` branch is selected (line 12). When the callee is a named person, the `private` branch is chosen (line 16). When the `ACK` request is received, the processing depends on whether the call is private or for the hotline. The correct branch is automatically selected.

2.5 Safety Properties

The telephony domain imposes stringent safety and robustness requirements. A service should not itself incur runtime errors and should respect the underlying protocol. We consider some kinds of errors that can occur when programming SIP services with existing SIP APIs, and show how SPL has been designed either to prevent these errors outright or to enable static verifications that detect these errors in SPL services.

Erroneous call processing A SIP service must ultimately perform some signaling action, such that no call is lost. Furthermore, the treatment of each message must be compliant with the SIP RFC 3261 [89]. For example, when a handler successfully forwards a message, it cannot then return an error. For another example, a handler is dedicated to a unique kind of request (*e.g.,* `INVITE`) and thus cannot forward a message of another kind (*e.g.,* `ACK`).

In SPL, signaling actions are associated with explicit keywords, such as `forward` for forwarding a request and `/SUCCESS` for matching a success response, allowing the SPL verifier to straightforwardly check that every execution path through a handler performs at least one signaling action, and that these signaling actions are coherent with each other. Furthermore, the SPL language prevents changing the method name when forwarding a request; the only argument to `forward` is the destination, leaving the request structure, and thus the method name, implicit.

A SIP message contains a number of headers, of which some are optional or read-only. Furthermore, as SIP messages are implemented as text, the underlying implementation of all headers is as strings, although a header may have a more intuitive meaning as *e.g.* an integer or a URI. Errors may include accessing a header that is not present in the current message, trying to modify the value of a read-only header, interpreting a header value as the wrong type of value, or writing the wrong type of value to a header. The SPL language prevents access to a header that is not present, via the use of the `when` construct that combines both a check for the presence of the header and access to the header value. SPL also provides the `with` construct for updating header values. In this case, the SPL verifier checks that uses of this construct only mention writable headers. Both constructs allow for headers to be treated as string-typed or to be accessed using more intuitive types; in the latter case the SPL verifier checks the validity of the type coercion.

Erroneous state and control management A SIP service typically does some initial processing of a request and then forwards the request to one or more parties. Typical SIP APIs separate this service logic into separate entry points for request and response processing. This strategy breaks up the treatment of a

given method, making it difficult to follow the service logic, and implies that state that is needed across the forwarding of a request must be saved and restored manually, a tedious and error-prone operation. In SPL, request and response processing are contiguous, within a single unit, with a forward operation being no more disruptive to the program structure than an ordinary procedure call. Local variables are implicitly saved and restored across a `forward`. Furthermore, SPL allows variables to be associated with an entire service, a registration, or a dialog, transparently managing the access to these variables across method invocations, thus ensuring that this data is manipulated in a consistent way.

Erroneous resource management SIP APIs based on general-purpose languages do nothing to protect against safety errors that can occur in these languages. For example, APIs do not protect against infinite loops, and APIs based on C do not protect against out-of-bounds accesses to data structures or accesses to freed data. SPL allows only bounded loops by the use of `foreach`, and includes appropriate checks on data structure accesses, as found in Java. Furthermore, SPL has no mechanism for dynamically allocating data, ensuring that service execution fits within a known memory bound.

2.6 Formal Semantics

We have formally specified the semantics of SPL, enabling a precise definition of its interaction with the SIP VM. This formal definition serves as a foundation for defining program analyses. In this section, we describe both the static and dynamic semantics of SPL, focusing on the semantics of sessions, method invocations, and forward expressions.

2.6.1 Static semantics

The purpose of the static semantics of a language is to specify what properties can be inferred about programs in that language, before execution. In the case of SPL, we present a static semantics that focuses on message forwarding. This static semantics can be enriched to consider other static properties, such as type safety.

In SPL, when a handler needs to forward a message, it uses the `forward` expression. This forwarding yields control to the SIP VM, which sends the request. When a positive response is received (*i.e.*, a `2xx` response code), no additional forwarding of the request is allowed and the response should be returned as the result of the handler. This restriction prevents a service from forwarding a call to someone other than the party who accepted it. We now present an analysis that ensures that this restriction is satisfied for any program written in SPL.

Abstract Syntax

For conciseness, we focus on a subset of SPL related to forward operations, noted SPL_{fwd}. The abstract syntax of SPL_{fwd} is presented in Figure 2.4. We only consider handlers that return a `response` value and declarations that introduce variables of type `response`.

Abstracting an SPL program into its SPL_{fwd} counterpart is straightforward. This is illustrated by the counter service previously shown in Figure 2.2, whose SPL_{fwd} counterpart is displayed in Figure 2.5. In addition, we assume that standard program transformations such as *copy propagation* and *constant propagation* are performed to obtain the SPL_{fwd} program.

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

Figure 2.4: Abstract syntax of the SPL_{fwd} language

```
response Some(incoming) INVITE {
  response r;
  r = fwd None;
  cond (r == /ERROR,
        r = fwd Some(...); return r,
        return r)
```

Figure 2.5: `INVITE` handler of the counter service in SPL_{fwd}

Semantic Rules

The semantic rules of SPL_{fwd} are described as inference rules with a sequence of premises above a horizontal bar and a judgment below the bar (see Figure 2.6). A judgment of the form $\tau_1 \vdash^D decs : \tau_2$ (rules 2.1 and 2.2) means that the evaluation of the declarations *decs* in the environment τ_1 returns a new environment τ_2. A judgment of the form $\tau_1 \vdash^S stmt : \langle \tau_2, \mathfrak{P} \rangle$ (rules 2.3 to 2.8) means that the evaluation of the statement $stmt$ in the context of the environment τ_1 returns the new environment τ_2 and the boolean value $\mathfrak{P}$. This value becomes false as soon as an illegal use of `forward` is detected.

A judgment of the form

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

(rule 2.9) means that the evaluation of the handler `response` $dir^?$ *method_name* `{` *D S* `}` returns the boolean value $\mathfrak{P}$. If $\mathfrak{P}$ is false, the handler contains an illegal use of `forward` and is rejected. If $\mathfrak{P}$ is true, the handler does not contain an illegal use of `forward` and is accepted.

A judgment of the form $\tau \vdash^{E_R} exp : \sigma$ (rules 2.10 to 2.12) means that the evaluation of the expression *exp* in the environment τ returns the status σ. The status is success if the expression is known to return a response code in the `2xx` class, error if the expression is known to return a response code not in the `2xx` class, and $\perp$ if the response code returned by the expression is not known. These status values form a partial order, with $\perp \sqsubseteq$ success and $\perp \sqsubseteq$ error. We use this ordering to determine how to merge environments in the static semantics of a conditional (rule 2.7).

Finally, a judgment of the form $\tau \vdash^{E_B} bool\_exp : (id, \sigma)$ (rule 2.13) means that the evaluation of a boolean expression in the environment τ returns a pair of an identifier *id* and a status value σ. The only boolean expressions allowed by SPL_{fwd} are equalities that compare an identifier to a description of a response, and thus evaluating such an expression to true or false gives information about the response code stored in the identifier. The information that can be obtained is then represented as a pair of the identifier *id* and the corresponding status value σ.

The key points of the analysis are in the treatment of `forward` (rules 2.3 and 2.4) and the treatment of conditionals (rule 2.7). Because the response received from a `forward` is always stored in a variable,

$$\frac{}{\tau \vdash^D \texttt{response}\ \ id : \tau[id \mapsto \mathsf{error}]} \quad (2.1) \qquad \frac{\tau \vdash^D D_1 : \tau_1 \qquad \tau_1 \vdash^D D_2 : \tau_2}{\tau \vdash^D D_1\ \ \texttt{;}\ \ D_2 : \tau_2} \quad (2.2)$$

$$\frac{\exists (x,\sigma) \in \tau.\ \sigma \neq \mathsf{error} \qquad \tau' = \tau[id \mapsto \bot]}{\tau \vdash^S id\ =\ \texttt{fwd}\ \ URI^? : \langle \tau', \mathsf{false} \rangle} \quad (2.3) \qquad \frac{\forall (x,\sigma) \in \tau.\ \sigma = \mathsf{error} \qquad \tau' = \tau[id \mapsto \bot]}{\tau \vdash^S id\ =\ \texttt{fwd}\ \ URI^? : \langle \tau', \mathsf{true} \rangle} \quad (2.4)$$

$$\frac{}{\tau \vdash^S \texttt{return}\ \ E_R : \langle \tau, \mathsf{true} \rangle} \quad (2.5) \qquad \frac{}{\tau \vdash^S \epsilon : \langle \tau, \mathsf{true} \rangle} \quad (2.6)$$

$$\frac{\begin{array}{c}\tau \vdash^{E_B} E_B : (id, \sigma) \\ \tau[id \mapsto \sigma] \vdash^S S_1 : \langle \tau_1, fwd_1 \rangle \\ \tau[id \mapsto \neg\sigma] \vdash^S S_2 : \langle \tau_2, fwd_2 \rangle\end{array}}{\tau \vdash^S \texttt{cond}\ \ (E_B\texttt{,}\ \ S_1\texttt{,}\ \ S_2) : \langle \tau_1 \uplus \tau_2, fwd_1 \wedge fwd_2 \rangle} \quad (2.7)$$

$$\frac{\begin{array}{c}\tau \vdash^S S_1 : \langle \tau_1, fwd_1 \rangle \\ \tau_1 \vdash^S S_2 : \langle \tau_2, fwd_2 \rangle\end{array}}{\tau \vdash^S S_1\ \ \texttt{;}\ \ S_2 : \langle \tau_2, fwd_1 \wedge fwd_2 \rangle} \quad (2.8)$$

$$\frac{\varnothing \vdash^D D : \tau_D \qquad \tau_D \vdash^S S : \langle \tau, fwd \rangle}{\vdash^H \texttt{response}\ \ dir^?\ \ method\_name\ \ \{\ \ D\ \ S\ \ \} : fwd} \quad (2.9)$$

$$\frac{(id, \sigma) \in \tau}{\tau \vdash^{E_R} id : \sigma} \quad (2.10) \qquad \frac{}{\tau \vdash^{E_R} \texttt{/SUCCESS} : \mathsf{success}} \quad (2.11)$$

$$\frac{}{\tau \vdash^{E_R} \texttt{/ERROR} : \mathsf{error}} \quad (2.12) \qquad \frac{\tau \vdash^{E_R} E_R : \sigma}{\tau \vdash^{E_B} id\ \ ==\ \ E_R : (id, \sigma)} \quad (2.13)$$

Figure 2.6: The static semantics of SPL_{fwd}

the environment τ is essentially a record of the effect of the `forward`s that have taken place. If there is any variable whose value is success or $\bot$, then some previous `forward` has or may have succeeded. In either case (rule 2.3) the `forward` is illegal and false is returned. Only if every variable has the value error (rule 2.4), do we know that all previous `forward`s have failed and the current `forward` is allowed. In both rules, the environment is updated by binding the identifier *id* to $\bot$, to indicate that the result of its `forward` has not been tested, and thus might be success. Finally, testing of the result of a `forward` takes place in a `cond` statement (rule 2.7), and uses rules 2.10 to 2.13. In rule 2.7, the "then" branch, S_1, is analyzed with respect to an environment reflecting the fact that the test is true, and the "false" branch, S_2, is analyzed with respect to an environment reflecting the fact that the test is false. The latter environment is constructed using the operator $\neg$, defined as $\neg\mathsf{success} = \mathsf{error}$, $\neg\mathsf{error} = \mathsf{success}$ and $\neg\bot = \bot$. The treatment of the statements S_1 and S_2 yields the two new environments τ_1 and τ_2. Because either may be available at execution time, the static semantics merges them using the operator $\uplus : \tau \times \tau \to \tau$, defined as follows:

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

This operator ensures that if the response for some `forward` is unknown in either branch, or is considered to have different values in the two branches, then the value of the corresponding variable becomes $\bot$.

2.6.2 Dynamic semantics

The purpose of a dynamic semantics is to specify what happens when a program is executed. We now present the dynamic semantics of SPL, examining the relationship between the language constructs and the underlying virtual machine. For conciseness, we ignore the branch construct. The complete dynamic semantics of SPL can be found here [77].

Sessions SPL is designed around the concept of a *session*, encapsulating a set of variables and handlers. To identify sessions uniquely, each session is associated with a unique *label*. To describe the position of a session in the hierarchy, a session is furthermore associated with an *address*, which is a sequence of the labels of the session and its ancestors. Information about a session is stored in a global state σ, mapping an address to a tuple containing some status information about the session, a *session environment* mapping the session variables to their values, and a list of the addresses of the sub-sessions:

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

The semantics of SPL uses the following set of functions that manipulate sessions: create_session, prepare_method_invocation, continue_session, and end_session.

The function create_session extends the global state with an entry for a new session and has the following type,

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

The entry's status information includes a flag true indicating that the session is live and a reference count 0 indicating that no handler is currently executing in the session. The entry's session environment is obtained by evaluating the session variable declarations in an environment binding the session variables of the ancestor sessions. Finally, the entry's list of sub-sessions is empty. The result is a new global state.

Once a session has been created, methods can be invoked within the session. Method invocation is initiated using the function prepare_method_invocation, of type:

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

This function extracts the declarations and body associated with the method, retrieves the sequence of session environments associated with the session and its ancestors, and increments the reference count, indicating that a handler is executing in the session. The declarations, body, and session environments are returned, with a new global state.

Execution of handler code manipulates the sequence of session environments of the current session and its ancestors, as obtained by prepare_method_invocation. When execution of the handler code completes, these environments must be reinserted into the global state, which is done by the function continue_session, having the following type:

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

The behavior of this function depends on whether the session is still live. If so, continue_session updates the global state with the new session environments and decrements the reference count, indicating that execution of the current handler has completed. If the session is no longer live, continue_session additionally calls end_session to determine whether the session should be destroyed.

Due to the sharing of state between handlers and between sessions, the most complex part of session management is session termination. Termination of a session is requested either when execution of certain handlers returns an error code or upon invocation of a final method (see Table 2.1). Nevertheless, it is not always desirable to destroy the session immediately. One issue is that handlers of the current session or its sub-sessions may be waiting for responses. For example, a dialog session can be waiting for a response to

a `REINVITE` request sent by one party when it receives a `BYE` request sent by the other party. When the `REINVITE` response arrives, some code may be executed by the `REINVITE` handler, which may refer to the session variables. Thus, a session is only destroyed when its reference count is 0, indicating that no handler is executing in the session. Another issue is that in some cases, a session may end from the point of view of SIP, but should persist at the SPL level. An example is a registration session, which terminates at the SIP level either on an explicit request or on expiration of a timer. Registration, however, is not necessary for existing dialogs or subscriptions to continue, and these dialogs or subscriptions may refer to the registration variables. Final methods are thus classified as *persistent*, meaning that the session no longer accepts sub-sessions but is not destroyed until all sub-sessions terminate, or *non-persistent*, meaning that the session and all sub-sessions terminate immediately, subject to the reference count constraint. For example, the method `unregister` for registrations is persistent, while the method `undeploy`, for services, is non-persistent to allow a system administrator to take down the system in a timely manner. Session termination is managed by the function end_session, which is invoked by continue_session whenever the session is not live.

Method invocation The semantics of SPL terms is described using a continuation-based abstract machine [1]. Execution in this machine is specified as a sequence of configurations, starting with a configuration representing the receipt of a message and ending with a configuration representing the returning of some information to the SIP VM. Intermediate configurations represent the execution of a term or the invocation of a continuation. A continuation is analogous to the stack used in the standard implementation of a procedural language. Whenever the semantics begins the execution of a term, it adds a frame to the continuation storing all of the information required to continue execution from the point of that term. This approach makes each configuration self-contained, and is used in the semantics of `forward`. The configurations used in the small step semantics of method invocation are as follows, where ϕ is the service code, *uri* is a pair of the destination of the request and an environment containing bindings derived from the message headers of the reques and s is a continuation:

- Method invocation: $\phi, \sigma \models_{\text{mi}} \langle \mathit{method\_name}(\mathit{address}), \mathit{direction}, \mathit{uri} \rangle$
- Method continuation: $\langle \sigma, \mathit{address} \rangle, \langle \mathit{envs}, \mathit{uri}, \mathit{local\_env} \rangle \models_{\text{mc}} s, \mathit{resp}$
- Handler body: $\langle \sigma, \mathit{address} \rangle, \mathit{envs}, \mathit{uri}, s \models_{\text{h}} \mathit{decls}, \mathit{stmt}$
- Handler statement: $\langle \sigma, \mathit{address} \rangle, \langle \mathit{envs}, \mathit{uri}, \mathit{local\_env} \rangle, s \models_{\text{s}} \mathit{stmt}$
- Return to the SIP VM: $\mathit{resp}_{\perp}, \sigma$

The semantic rules are described as inference rules, with a sequence of premises above a horizontal bar, and the current configuration and the next configuration in the execution sequence below the bar, separated by an arrow. We focus on dialog methods. The treatment of other kinds of methods is similar.

An initial `INVITE` method creates a new dialog session:

$$\frac{\begin{array}{c}\mathsf{create\_session}(\phi, \sigma, \mathit{address}, \texttt{undialog}) = \sigma' \\ \mathsf{prepare\_method\_invocation}(\phi, \sigma', \texttt{INVITE}, \mathit{direction}) = \langle \mathit{decls}, \mathit{stmt}, \mathit{envs}, \sigma'' \rangle\end{array}}{\phi, \sigma \models_{\text{mi}} \langle \texttt{INVITE}(\mathit{address}), \mathit{direction}, \mathit{uri} \rangle \Rightarrow \langle \sigma'', \mathit{address} \rangle, \mathit{envs}, \mathit{uri}, \langle \texttt{INV}\ \ \phi \rangle \models_{\text{h}} \mathit{decls}, \mathit{stmt}}$$

$$\frac{\mathsf{continue\_session}(\phi, \sigma, \mathit{address}, \mathit{envs}) = \sigma'}{\langle \sigma, \mathit{address} \rangle, \langle \mathit{envs}, \_, \_ \rangle \models_{\text{mc}} \langle \texttt{INV}\ \ \phi \rangle, \texttt{/SUCCESS/}\mathit{resp} \Rightarrow \texttt{/SUCCESS/}\mathit{resp}, \sigma'}$$

$$\frac{\begin{array}{c}\mathsf{set\_persistence}(\sigma, \mathit{address}, \mathsf{false}) = \sigma' \\ \mathsf{continue\_session}(\phi, \sigma', \mathit{address}, \mathit{envs}) = \sigma''\end{array}}{\langle \sigma, \mathit{address} \rangle, \langle \mathit{envs}, \_, \_ \rangle \models_{\text{mc}} \langle \texttt{INV}\ \ \phi \rangle, \texttt{/ERROR/}\mathit{resp} \Rightarrow \texttt{/ERROR/}\mathit{resp}, \sigma''}$$

The first rule initiates the method invocation by creating the session and extracting the handler code and relevant session environments. The rule produces (bottom line) a configuration causing execution of the handler code. The continuation in this new configuration is labeled `INV`, indicating that after executing the handler body, some work should be done that is specific to an `INVITE` method. The second and third rules describe the invocation of this continuation. In the second rule, the result of the handler is a success code, in which case it only remains to update the global state using continue_session. In the third rule, the result of the handler is an error code, in which case set_persistence is called to indicate that the session is no longer live and that its termination should be nonpersistent (indicated by false). These changes to the session status cause the subsequent call to continue_session to call end_session to destroy the session.

Invocation of a medial or final dialog method is similar, except without the use of create_session. At the end of a medial method, the session always continues, and thus the continuation rule for a medial method is analogous to the `/SUCCESS` rule for the `INV` continuation. At the end of a final method, the session always terminates, and thus the continuation rule for a final method is analogous to the `/ERROR` rule for the `INV` continuation. While the termination of a session due to an error code in an initial method is always nonpersistent, the termination of a session due to invocation of a final method depends on the persistence associated with the method itself. The third argument to set_persistence is thus adjusted accordingly.

Method body Evaluation of the body of a hander starts by creating an environment with the values of local variables, as described by the following rule.

$$\frac{\langle envs, uri, \emptyset\rangle \models_{\text{d}} decls : local\_env \qquad \rho = \langle envs, uri, local\_env\rangle}{\tau, envs, uri, s \models_{\text{h}} decls, stmt \Rightarrow \tau, \rho, s \models_{\text{s}} stmt}$$

Expressions We consider a simplified set of expressions, as shwon Figure 2.4, in in which there is only identifiers, constants, and one binary operator for equality testing. The semantics of expression uses the following configurations:

- Expression: $\langle \sigma, address\rangle, \langle envs, uri, local\_env\rangle, s \models_{\text{e}} exp$
- Expression continuation: $\langle \sigma, address\rangle, \langle envs, uri, local\_env\rangle \models_{\text{ec}} s, value$
  value is an arbitrary value.

Equality is associated with two continuations : $(\texttt{EQ1}\ exp_2)$ and $(\texttt{EQ2}\ v_1)$. The first continuation stores the second expression to evaluate. The second continuation stores the result of the evaluation of the first expression, so as to compare it with the result of the evaluation of the second expression. The semantic rules for expression are as follows:

$$\begin{aligned}
\tau, \rho, s \models_{\text{e}} x \quad &\Rightarrow \quad \tau, \rho \models_{\text{ec}} s, \rho(x) \\
\tau, \rho, s \models_{\text{e}} cst \quad &\Rightarrow \quad \tau, \rho, (\texttt{EQ1}\ exp_2) :: s \models_{\text{e}} exp_1 \\
\tau, \rho \models_{\text{ec}} (\texttt{EQ1}\ exp_2) :: s \quad &\Rightarrow \quad \tau, \rho, (\texttt{EQ2}\ v_1) :: s \models_{\text{e}} exp_2 \\
\tau, \rho \models_{\text{ec}} (\texttt{EQ2}\ v_1) :: s, v_2 \quad &\Rightarrow \quad \tau, \rho \models_{\text{ec}} s, v_1 == v_2
\end{aligned}$$

Statements The semantics of statements uses the following configuration:

$$\langle \sigma, address \rangle, \langle envs, uri, local\_env \rangle \models_{sc} s, resp_{\perp}$$

In this configuration, $resp_{\perp}$ is either a response or $\perp$ when no response is returned as it is the case for statements other than `return` and `forward`. The following semantic rules use `SEQ`, `RETURN`, `FORWARD` and `COND` to dentote the continuations for sequence, return, forward and conditional statements.

$$\tau, \rho, s \models_{s} S_1 \texttt{ ; } S_2 \Rightarrow \tau, \rho, (\texttt{SEQ } S_2) :: s \models_{s} S_1$$

$$\tau, \rho, s \models_{s} \texttt{return } exp\texttt{;} \Rightarrow \tau, \rho, (\texttt{RETURN}) :: s \models_{e} exp$$

$$\tau, \rho, s \models_{s} \texttt{cond(}E\texttt{, } S_1\texttt{, } S_2\texttt{)} \Rightarrow \tau, \rho, (\texttt{COND } S_1 \ S_2) :: s \models_{e} exp$$

$$\tau, \rho \models_{ec} (\texttt{COND } S_1 \ S_2) :: s, \mathsf{true} \Rightarrow \tau, \rho, s \models_{s} S_1$$

$$\tau, \rho \models_{ec} (\texttt{COND } S_1 \ S_2) :: s, \mathsf{false} \Rightarrow \tau, \rho, s \models_{s} S_2$$

$$\tau, \rho \models_{ec} (\texttt{RETURN}) :: s, resp \Rightarrow \tau, \rho \models_{sc} s, resp$$

$$\tau, \rho \models_{sc} (\texttt{SEQ } S_2) :: s, \_ \Rightarrow \tau, \rho, s \models_{s} S_2$$

A statement can reinvoke the underlying machine via a `forward` operation. For this, we have to bring ρ up to date with respect to the changes that have occured in *envs*. When control returns to the semantics, we similary have to obtain the new *envs*. For these operations, we have to keep the environments separated. In the following, we consider only the case where `forward` has no arguments, as illustrated in line 18 of Figure 2.2. The rules for the other case are similar.

$$\frac{\mathsf{update\_envs}(\sigma, address, envs) = \sigma'}{\langle \sigma, address \rangle, \langle envs, uri, local\_envs \rangle, s \models_{s} x = \texttt{forward} \Rightarrow \mathsf{forward}(uri, (\texttt{FORWARD } address \ uri \ local\_env) :: (\texttt{ASSIGN } x) :: s), \sigma'}$$

$$\frac{\mathsf{lookup\_envs}(\sigma, address) = envs}{\mathsf{forward\_response}(resp, (\texttt{FORWARD } address \ uri \ local\_env) :: s), \sigma \Rightarrow \langle \sigma, address \rangle, \langle envs, uri, local\_env \rangle \models_{sc} s, resp}$$

$$\tau, \rho \models_{sc} (\texttt{ASSIGN } x) :: s, v \Rightarrow \tau, \mathsf{update}(x, v, \rho) \models_{sc} s, \perp$$

The first rule initiates the forwarding operation. This rule uses update_envs to update the global state with the current values of the session variables. Then, it calls the function forward to pass the current continuation `s`, as well as the address of the session and the values of local variables, to the SIP VM, which forwards the message. The second rule describes what happens when the response arrives, as indicated by the presence of a forward_response configuration. In this case, the SIP VM passes the response and the continuation used when calling forward back to SPL, which reconstructs the environment and applies the continuation to the response in order to continue the handler execution. The last rule updates the value of x in the environment ρ with the result of the previous forward operation and continues the execution of the handler.

2.7 Related Work and Assessment

Various approaches have been proposed to develop SIP-based telephony services. The SIP Express Router platform [52] relies on a restricted configuration language to define the message routing logic. Its API also offers hooks to extend the core platform with modules written in C. However, analyzing such modules to ensure that a service respects the SIP protocol automaton can be very challenging. The Microsoft Live Communications Server [73] introduces a dedicated language for coarse-grained dispatch of SIP messages. Services that require advanced functionalities can shift the processing of a message to a C# program that can access the platform through a powerful API. Consequently, programmers must choose between expressiveness and simplicity. JAIN SIP [74] and SIP Servlet [53] are the standard Java interfaces to a SIP signaling stack. They provide a powerful solution for developing SIP services. However, programmers still have to deal with details of the protocol.

High level scripting languages such as CPL [61], LESS [100] and CCXML [97] have emerged for developing SIP services. Some approaches have been proposed to verify properties of services written in these languages. For example, detection of interaction between features has been explored in the context of CPL [76, 102] and LESS [101]. However, none of these languages has been formally defined. Therefore, language implementations and verifications rely on informal specification found in the available documentation, making them subject to variation.

The formal definition of SPL serves not only as documentation but also as a foundation for implementing the language and its run time. Our first prototype implementation of the SPL interpreter was implemented in OCaml by a single developer in about one week. This implementation is a straightforward mapping of the dynamic semantics, where each inference rule is translated into an OCaml function. The complete semantics of SPL consists of about 100 inference rules. Our implementation of the SPL interpreter written in OCaml is very close in size. In addition to this first prototype, a developer has used the SPL semantics as a reference to implement a Java version. While being much more verbose, the Java version only required two weeks to develop. The implementation of the SIP VM presented in Section 2.3 has been developed in Java and is based on a JAIN-SIP stack [37].

The prototype implementation of the SIP VM and the SPL interpreter have been integrated to the SIP-based telephony system of our university, and various services have been written in SPL. In addition to traditional services such as black listing or conditional redirections, SPL has proved its usability to define advanced services. For example, an SPL service has been deployed that allows a secretary to manage incoming calls, using a waiting queue, depending on a shared agenda, availability of the personnel, and information about the call such as the name of the caller, if known. In these experiments, SPL has demonstrated its usability and ease of programming. In addition, our implementations based on the formal semantics described in Section 2.6 have proved their robustness while showing no significant performance penalty.

2.8 Conclusion

In this chapter, we have presented a domain-specific virtual machine for SIP, providing the programmer with a high-level interface dedicated to telephony service development. This SIP VM is centered around the notion of a session that structures the development of a service. Additionally, we have introduced a DSL named SPL that offers high-level notations and abstractions for service development. This language hides the subtleties of SIP platforms, making programs more concise than their GPL counterparts, without sacrificing expressivity.

Several languages have been proposed to make SIP-based service creation easier and faster. However, they are only defined informally, which makes it difficult for developers to understand the subtleties of the language or to implement the language and run-time systems. In contrast, SPL is formally defined, making explicit in a high-level way an expertise in SIP-based telephony service creation and a model for run-time systems. In our experience, this repository of knowledge has proved to be extremely valuable for

porting SPL from OCaml to Java. Furthermore, formalizing SPL has made it possible to thoroughly cover both language and run-time design issues, prior to implementing the language. The static and dynamic semantics of SPL enable a precise definition of its interaction with the SIP VM. The formal definition of the SIP VM and SPL is a foundation for defining program analyses and serves as a documentation for both service programmers and platform developers.

A variety of services have been written in SPL for our university department. In these experiments, SPL has demonstrated its usability and ease of programming. Its robustness has been a key factor in expediting service deployment.

Chapter 3

Zebu: A Language-Based Approach for Network Protocol Message Processing

This chapter presents Zebu [21, 22], a domain-specific language to ease the development of protocol-handling layers. Zebu specifies the protocol-handling layer of network applications that use textual HTTP-like application protocols. The Zebu language is based on the ABNF formalism used for protocols specifications and enables fine tuning the protocol-handling layer according to the needs of a particular application. We demonstrate, through various experiments, that this approach is a reliable alternative to manual development in the context of protocol-handling layers. We show that Zebu generated code has good performance and has a significantly lower memory footprint than comparable existing manually encoded solutions.

3.1 Introduction

In the Internet era, many applications, ranging from instant messaging clients and multimedia players to HTTP servers and proxies, involve processing network protocol messages. This processing is performed by the *protocol-handling layer* of a network application, which must both parse messages as they are received from the network and craft new messages to be sent onwards. As the protocol-handling layer represents the front line of interaction between the application and the outside world, its correctness is critical; any bugs can leave the application open to attack [85]. In the context of in-network applications such as proxies, where achieving high throughput is essential, both message parsing and the crafting of new messages must be very efficient.

In this chapter, we concentrate on protocol-handling layers for HTTP-like protocols. These protocols are text-based and line-oriented, and include the Session Initiation Protocol (SIP) [89], for telephony over IP, the Real Time Streaming Protocol (RTSP), for negotiating multimedia parameters, and the Single Mail Transfer Protocol (SMTP) [83], for email transfer over the Internet. HTTP-like protocols are widely used, to provide interoperability between diverse applications and devices, and because the text representation makes debugging easy. They are also easily extensible. For example, more than 150 extensions to the SIP protocol have been proposed. Thus, it is often not sufficient to simply develop one protocol-handling layer for a given protocol and reuse it in all applications. Instead, protocol-handling layers must be continually developed and extended to take into account new protocol variants.

Implementing a correct and efficient protocol-handling layer for an HTTP-like protocol, however, is a difficult task. The message structure for such a protocol is typically defined in a "Requests for Comments" (RFC), using an extended form of BNF with English text specifying various properties of the message values. A BNF specification amounts to a state machine, which for efficiency is often implemented in an unstructured way using *gotos*, as exemplified by the SIP and RTSP parsers considered in our evaluation

(Section 3.7). The resulting code is thus error-prone and hard to maintain, *e.g.*, when a protocol extension needs to be supported. Because of the need to maximize throughput, performance is also a significant issue. One common approach to improve performance is to process message fragments only as they are needed, allowing the parser to skip fragments that are not needed for a given application. For example, a router normally only uses the message elements that describe the message destination [58]. However, when the skipped fields turn out to be needed in specific situations, complex parsing code may end up scattered throughout the application. Finally, HTTP-like protocols are increasingly used in embedded systems, to enable interoperability with other devices and to allow the reuse of existing network tools [56, 60, 94]. In addition to the need for efficient performance, embedded systems typically bring tight space constraints.

In the compiler construction community, parsers have long been constructed using automated parser generators such as `yacc` [54]. Nevertheless, such tools are not suitable for generating parsers for use in a protocol-handling layer, as the grammars provided in RFCs are often not context free, due to additional constraints present in the RFC text, and such tools provide no support for deferring the parsing of some message fragments. Thus, protocol-handling layers have traditionally been implemented by hand. This situation, however, is becoming increasingly impractical, given the variety and complexity of protocols and their extensions. Incorrect or inefficient parsing makes an application vulnerable to denial of service attacks, as illustrated by the "leading slash" vulnerability found in the Flash HTTP Web server [85].

This chapter

In this chapter, we propose to address the issues of correctness and efficiency of network protocol message processing code using a parser-generator-based approach. For this, we present a domain-specific language, Zebu, for describing HTTP-like text-based protocol message formats and related processing constraints. Zebu amounts to an annotated version of ABNF[1] (*Augmented BNF* [34]), the variant of BNF used in RFCs. Using Zebu, developing a textual protocol network parser consists of copying the ABNF grammar found in the protocol specification and then adding some annotations to tailor the parser to application needs. Zebu complements ABNF with annotations indicating which message fields should be stored in the data structures provided to the application, and other semantic information. Fields can be declared as `lazy`, which gives the application control over the time when the field is parsed. A Zebu specification is processed by a compiler that translates the ABNF rules into regular expressions implemented using the PCRE library [50]. The result of this compilation is a set of stub functions that are to be used by an application to process network messages. Based on the annotations, this compiler implements domain-specific optimizations to reduce the memory usage of a Zebu-based application. Besides efficiency, Zebu also addresses robustness, as the compiler performs many consistency checks, and can generate parsing stubs that validate the message structure.

Contributions

The contributions of Zebu are as follows:

Ease of use. Zebu can generate a parser directly from an RFC ABNF specification, with only a few modifications to distinguish between requests and responses and to highlight certain parser entry points. Auxiliary constraints on message values, such as the bounds on integer values or constraints on the relationship between the values of multiple fields, as are typically found in the RFC text, can be added as annotations within the ABNF specification provided to Zebu.

Ease of integration with applications. Annotations can be added to the ABNF specification to indicate the message elements that are of interest to a given application. The Zebu compiler generates stubs that

[1] Currently, the implementation of Zebu relies on the PCRE library [50], implying that the ABNF must be limited to what can be expressed using regular expressions (see Section 3.4.4).

provide direct access to this information. Further annotations can be used to specify the types to be used for specific message element values, allowing these stub functions to convert these values from their string representation in the message to the data type that is most convenient for the application. This conversion also performs any entailed data consistency checks (*e.g.*, a value specified to be a 16 bit integer is checked to have a value that fits within 16 bits). Finally, Zebu provides facilities for easing the integration of a protocol-handling layer into an embedded system.

Robustness. Because Zebu parsers are based directly on the ABNF, they can be very robust. In experiments involving messages known to be difficult to parse, we show that protocol-handling layers based on Zebu detect 100% of erroneous received messages as opposed to 25% for existing, widely used, manually developed parsers.

Efficiency. Zebu incorporates a number of state-of-the-art strategies for efficiently parsing and crafting messages for HTTP-like protocols, thus relieving the developer of the need to identify and hand code these strategies. These optimizations address not only run-time performance but also memory usage. Experiments have shown that Zebu-generated protocol code and hand-written code with equivalent levels of robustness have the same performance.

Outline

The rest of this chapter is organized as follows. Section 3.2 provides some insights about network applications, their structure and the kind of programming they entail. Section 3.3 presents some previous work on improving the protocol-handling layer development process. Sections 3.4 and 3.5 describe the Zebu language, its implementation, and its integration into the development of a network application. The rest of the chapter assesses Zebu in terms of robustness and performance. Finally, Section 3.8 concludes the chapter with some final remarks.

3.2 Protocol-Handling Layer

We focus on network processing at the application level, and more specifically on its lowest part, which we refer to as the *protocol-handling layer*. This layer enables communication between applications through application-centric protocols. It must take into account both the constraints of the protocol, which determine the message structure, and the requirements of the application, which determine how the message elements will be used. Typically, the protocol-handling layer amounts to 15-25% of the application code size and consumes 25% of the total message processing time [32, 31, 98]. This layer must thus be robust, efficient and maintainable/extensible. Combined with the inherent complexity of network protocol message formats, these three issues make the development of the protocol-handling layer very difficult.

3.2.1 Protocol Requirements

The protocol defines the structure of messages, and thus the definition of the message parser and the facilities for crafting new messages. The parser must receive incoming messages from the network, check their validity, and make the various message elements available to the application. The facilities for crafting new messages must in turn receive the data to transmit from the application and convert it to a format that respects the message structure defined by the protocol.

In developing a parser, the main difficulty is the large gap between the high level formalism, ABNF, used to specify protocols and the corresponding low-level code implemented by developers, typically in C, for efficiency. This gap is illustrated in Figure 3.1, which represents the ABNF grammar of one crucial SIP

message element, the `Via` header, and Figure 3.2, which represents the corresponding code extracted from the SIP Express Router (SER) [58]. SER is a SIP proxy whose protocol-handling layer is considered as the reference in terms of protocol support, both in terms of robustness and efficiency. The parsing code for the `Via` header is implemented as a state machine involving 64 *switches*, 564 *cases*, 232 *breaks* and 143 *gotos*. Although efficient, such low-level code has major potential pitfalls; notably, overlooking a single instruction can corrupt the whole protocol-handling layer and lead to security vulnerabilities. The code is essentially unreadable, and is difficult to maintain and extend.

```
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 =  name SLASH version SLASH transport
name =  "SIP" / token
version =  token
transport =  "UDP" / "TCP" /
             "TLS" / "SCTP" / other-transport
sent-by =  host [ COLON port ]
ttl =  1*3DIGIT ; 0 to 255
```

Figure 3.1: ABNF extract for the VIA header

```
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_BRANCH:
        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;
[...]
```

Figure 3.2: VIA header parsing in SER

An additional difficulty in implementing a parser is that the protocol-handling layer must detect messages that are invalid according to the protocol specification and respect the expected behavior with valid messages. In practice, according to a recommendation from the IETF, protocol-handling layers should accept messages that are almost valid. However, if this permissiveness is not controlled, it can lead to security holes. For instance, one could take control of a machine hosting the virtual PBX Asterisk [92] by sending particular erroneous SIP messages.[2]

In developing code for crafting new messages, the main difficulty is that a network message for an HTTP-like protocol consists of a single string amounting to the concatenation of various message elements, not all of which may be provided by the application at the same time. Careful coding is thus required to

[2] Asterisk SIP DoS vulnerability (empty REGISTER):
http://www.securiteam.com/unixfocus/5VP0C0UKUO.html

accumulate this information as it becomes available and then to use it to construct a complete message, without too much memory allocation and copying. The latter is particularly significant in the case of embedded systems where memory resources may be scarce.

3.2.2 Application Requirements

Applications differ in the means by which they request message elements, the format in which they expect to receive the message data, and the elements of the message they need to be informed about. We examine these application requirements in turn.

There are basically three ways for an application to interact with the protocol-handling layer: event-driven, black-box and delegate. In an event-driven model, the protocol-handling layer raises events when it has identified a subpart of a message and these events are then caught by the application logic. The application logic must keep track of what has been parsed. The black-box approach considers the protocol-handling layer as an opaque component. The message is only manipulated through a refined view made accessible by stubs. Getter and setter stubs enable getting and modifying information within a message. Composer stubs enable crafting new messages. In this case, the application logic is unaware of the data layout and of what has actually been parsed. The last approach is to provide the application logic with full access to the message representation. The protocol-handling layer waits until it has received the whole message to build a data structure that is then hoisted to the application logic. The application logic then has full access to the message elements.

When the protocol-handling layer makes available a message element to the application logic, it can provide the information about the message according to a more or less refined view. A *raw* presentation is a flat view consisting of the sequence of tokens found in the message. A *hierarchical* presentation groups the tokens of each message subpart into a separate data structure. Finally, a *logical* presentation includes some interpretation of the message elements, for instance with each kind of message element being represented by its proper application-level type.

The information about a message required by the application logic may differ according to the nature of the application being implemented. For instance, a web proxy may only need to look at a few message elements to perform some sanity checks and then at the actual destination to perform its task. On the other hand, a proxy such as Squid [99] that uses caching needs some additional information such as the object associated with the requested resource and some time-stamps. Finally, protocol gateways, translating messages from a source protocol towards a target one, examine only the message elements that can be projected into the target protocol.

In the case of Zebu, we target applications using the black box approach, to which the parser provides a logical view of the message, and where the parser can be constrained to only parse the fields that are relevant to the application. These choices reduce the programming burden on the application developer, by encapsulating complicated parsing and analysis in the generated code, while eliminating unnecessary work in the parser, thus providing efficiency.

3.3 Related Work

This section presents some existing approaches that have the goal of easing the development of the protocol-handling layer, while addressing the challenges of robustness, efficiency, maintainability and extensibility.

The domain-specific language PADS [43] provides a generic approach to easing data processing, targeting semi-structured *ad hoc* data, such as web server logs. A PADS description is a sequence of type declarations similar to those found in C, plus some semantic constraints on the data. This approach is, however, ill suited for developing network application protocol support. First, the language is very far from the ABNF notation, and thus requires a potentially complex specification translation phase. Notably, some elements of textual protocols, such as URLs, cannot be expressed precisely in terms of standard types and

thus must be represented using regular expressions. Regular expressions may in practice be substantially more complicated than ABNF, as illustrated later in this chapter with an extract of the regular expression for URLs (Figure 3.10). Furthermore, the main objective of PADS is to be able to process data records of several gigabytes (it was indeed developed to process the logs of an Internet Service Provider); not to react to small, high-throughput messages. Hence, it has a significant performance penalty, as illustrated in Section 3.7.1, that can be neglected when performing complex processing on huge amounts of data but is prohibitive in our context.

PacketTypes [70] is a domain-specific language dedicated to supporting binary protocols. It relies on type analysis to eliminate the need for writing all the low-level C code required for the analysis of received messages, with at least a 15% performance penalty. It does not support textual protocols.

Binpac [79] is a language integrated in the Network Intrusion Detection System (NIDS) Bro [81]. A NIDS verifies each received message to detect malicious ones. The goal of Binpac is to ease this analysis by defining the message structure. Like PADS, Binpac is also based on type descriptions. Hence, a substantial translation phase is again required to use it, including the use of regular expressions, which compromises the safety of the generated parsers. Furthermore, the current implementation of Binpac is tightly-coupled with the Bro implementation preventing its use as an independent tool.

The GAPA project and its associated language GAPAL [10] also aim to generate application-level protocols analyzers in NIDS. GAPAL is the first language for packet description based on ABNF. Like Zebu, it permits annotating an ABNF specification with types, to specify the kind of value that should be created from a message element, and labels, for referring to the message elements. However, the work on GAPAL does not consider implementation and usability issues, such as limiting the memory usage, providing incremental parsing, or facilitating the creation of new messages. The parsing code is to be used within the GAPA system, and thus no stubs are generated to allow interaction with an arbitrary application. Finally, GAPAL focuses on the parsing of incoming messages, while Zebu also provides features, such as lumps (see Section 3.5.3), for efficient modification of received messages before they are resent over the network. Therefore, Zebu addresses the full spectrum of network message processing, from message parsing to message creation.

Melange [69] is an architecture to implement protocols that integrates formal methods to improve server robustness: 1) ML-like static type systems; 2) model checking to verify the safety of the servers; and 3) generative meta-programming to express high level constraints for domain-specific tasks such as packet parsing and state-machine definition. The compiler generates a parser in OCaml which is both efficient and type-safe. However, this solution is too monolithic to be easily integrated in legacy programs.

3.4 The Zebu Language

To address the difficulty of implementing protocol-handling layer code, we have developed the domain-specific language Zebu. The main objective of the Zebu language is to allow the developer to specify the message syntax using a high-level notation, while minimizing the need for developer intervention in the complex process of translating this notation to C code. Accordingly, the Zebu language is based on the ABNF notation used in RFCs to specify the syntax of protocol messages. The developer only needs to slightly refactor the ABNF specification to isolate message elements specific to requests or responses. He can also optionally annotate the different message elements that should be exposed to the application. The Zebu compiler then generates a complete protocol-handling layer tailored to the application according to the provided annotations. In this section, we first describe the structure of HTTP-like protocol messages and present the ABNF notation, and then consider how an ABNF specification is converted to a Zebu specification, reflecting both protocol-related and application-related aspects. The complete grammar of Zebu is provided in Appendix B.

3.4.1 HTTP-like protocol messages and ABNF

A message of a HTTP-like protocol is structured according to a fixed line-based pattern. The first line, known as the *command* line, identifies the message as either a request or a response. Subsequent lines contain a sequence of *message headers*. Each header consists of a key followed by a number of field values, which define various properties of the communication. Finally, a message can optionally contain an arbitrary payload. Figure 3.3 shows a simple SIP INVITE request message, which illustrates these features. This fixed message structure underpins the design of the Zebu language.

```
INVITE sip:bob@biloxi.cs.com SIP/2.0
Via: SIP/2.0/TCP client.ny.cs.com:5060;branch=z9hG4bK7
Max-Forwards: 70
Route: <sip:ss1.ny.cs.com;lr>
From: Alice <sip:alice@ny.cs.com>;tag=9fxced76sl
To: Bob <sip:bob@biloxi.cs.com>
Call-ID: 3848276228511@ny.cs.com
CSeq: 1 INVITE
Contact: <sip:alice@ny.cs.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
[...]
```

Figure 3.3: SIP message, adapted from the SIP RFC

Figure 3.4 illustrates how the HTTP-like message structure is expressed in ABNF, for the SIP protocol, focusing on the rules relevant to INVITE requests. The first rule (line 1), expresses the possibility that a message can either be a request or a response. If it is a request, it consists of a request line, followed by a sequence of message headers, followed by a blank line, and then a payload (line 2). Line 3 defines the structure of a request line, comprising not only tokens, as derived by *e.g.* the nonterminal `Method`, but also whitespace, as derived by `SP` and `CRLF`. `Method` is defined as an alternation, as indicated by `/`, of the names of the standard SIP methods, as well as the nonterminal `extension-method`, allowing for extensions. The name of the INVITE method is specified in line 6, in ASCII. This ensures that INVITE is only recognized in uppercase; other strings such as `"SIP"` in line 5 are case-insensitive. Line 9 defines the CSeq header, which is found in INVITE and other requests. This header consists of the key `"CSeq"`, which is case insensitive, immediately followed by a colon, one or more digits, arbitrary whitespace, and a method name. Line 10 defines the To header, which is structured similarly.

```
1 SIP-message = Request / Response
2 Request = Request-Line *( message-header ) CRLF
              [ message-body ]
3 Request-Line = Method SP Request-URI SP SIP-Version CRLF
4 Request-URI = SIP-URI / SIPS-URI / absoluteURI
5 SIP-Version =  "SIP" "/" 1*DIGIT "." 1*DIGIT
6 INVITEm = %x49.4E.56.49.54.45 ; INVITE in caps
7 Method = INVITEm / ACKm / OPTIONSm / BYEm
              / CANCELm / REGISTERm / extension-method
8 extension-method = token
9 CSeq = "CSeq" HCOLON 1*DIGIT LWS Method
10 To = ( "To" / "t" ) HCOLON ( name-addr / addr-spec )
              *( SEMI to-param )
11 LWS =  [*WSP CRLF] 1*WSP ; linear whitespace
12 HCOLON =  *( SP / HTAB ) ":" SWS
```

Figure 3.4: Selected lines of the SIP ABNF (RFC 3261)

For many protocols, the ABNF specification does not completely define the message structure. Further constraints on the message structure are found in the accompanying text. Typically, the text in an RFC

defines when a header can appear, whether a header is mandatory, and how many times it can appear. For example, the SIP RFC specifies that the `Via` header can appear multiple times. Header field values may be required to be within a certain range or to have the same value as other message elements. For example, the SIP RFC specifies that the `Method` field in the `CSeq` header must be the same as the method specified in the request line. Finally, for message crafting, the RFC can identify which headers are read-only, constant parameters for the whole communication, and thus must be preserved unchanged in new messages.

3.4.2 Protocol-related aspects

Figure 3.5 shows a minimal Zebu specification derived from the lines of the SIP ABNF presented in Figure 3.4 (keywords are in bold face). The `message` block defines the scope of the whole specification. Because some message elements can only occur in either a request or a response, there is one block per message kind: message elements specific to requests are contained in a `request` block and message elements specific to responses are contained in a `response` block. The remainder of a Zebu specification is essentially the same as the ABNF specification of the message syntax.

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

Figure 3.5: A minimal Zebu specification, based on the SIP ABNF lines in Fig. 3.4

The refactoring of an ABNF specification into a Zebu specification is quite simple and systematic. The Zebu programmer defines the syntax of a request or response command line, minus the final newline, with the `requestLine` or `statusLine` keyword, respectively.[3] Hence, for instance, in Zebu, the ABNF rule for the `Request-Line` non-terminal (Figure 3.4, line 3) becomes:

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

The `header` keyword is then added ahead of each header definition. For a header, the key description (the header *name*) is moved from the right-hand side to the left-hand side, replacing the rule identifier. Hence, the Zebu rule reflects the key/value structure of an header. For instance, the *CSeq* ABNF rule (Figure 3.4, line 9) is reorganized into the following Zebu rule (Figure 3.5, line 14):

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

The `HCOLON` delimiter following the key is dropped to reduce clutter, since it appears in this position in all HTTP-like protocols. Zebu can, however, be configured to use another character, such as '=', as a delimiter. Some headers, such as the `To` SIP header, can be represented by several keys, typically various abbreviated versions. In this case, the name of the header is followed by the set of associated keys between braces, as illustrated in Figure 3.5, line 15. As in ABNF, key names and any other strings specified in a Zebu specification are case-insensitive.

[3] Zebu targets line-oriented protocols, and thus `CRLF` is assumed to be newline. This assumption enables efficient line-oriented parsing, as described in Section 3.4.3.

Once the minimal Zebu specification is created from the ABNF, annotations can be added to represent the header constraints found in the RFC text, as illustrated in Figure 3.6. These annotations appear between braces at the end of the corresponding grammar rules. Atomic constraints notably include `mandatory` for a header that must be present (*e.g.*, the `To` header, Figure 3.6, lines 27–29). More complex constraints can be expressed using C-like conditional expressions. For instance, Figure 3.6, line 4 specifies that the methods specified in the request line and the CSeq header must be identical. Some header constraints are request or response specific. These constraints are specified in the corresponding block. In the SIP specification, the `request` block (Figure 3.6, lines 2–6) requires, among other constraints that the `Max-Forwards` header be present, and the request line and CSeq header methods be equal. The response block constraints have been elided.

```
1 message sip3261 {
2  request {; Request only
3   requestLine = Method:method SP Request-URI:uri SP Version
4   header CSeq {CSeq.method == requestLine.method}
5   header Max-Forwards {mandatory}
6 [...]
7  }
8  response {; Response only
9   statusLine =
10       Version SP Status-Code:code SP RPhrase:rphrase
11 [...]
12  }
13 struct Request-URI =
14           SIP-URI / SIPS-URI / absURI {lazy}
15 INVITEm = %x49.4E.56.49.54.45 ; INVITE in caps
16 enum Method = INVITEm / ACKm / OPTIONSm / BYEm
17      / CANCELm / REGISTERm / extension-method
18 extension-method = token
19 uint16 Status-Code = Informational / Redirection
20      / Success / Client-Error
21      / Server-Error / Global-Failure / extension-code
22 uint16 Global-Failure = "600"/ "603"/ "604"/ "606"
23 uint16 extension-code = 3DIG
24      {extension-code>=100 && extension-code<=699}
25 header CSeq = 1*DIG:number as uint32 LWS Method:method
26 header Max-Forwards = 1*DIG:value as uint32 {mandatory}
27 header To {"to" / "t"} =
28      (name-addr / addr-spec:uri) *( SEMI to-param )
29      {mandatory, ReadOnly}
30 name-addr = [display-name] LAQUOT addr-spec:uri RAQUOT
31 struct addr-spec = SIP-URI / SIPS-URI / absURI {lazy}
32 [...]
```

Figure 3.6: Excerpt of a SIP Zebu specification

3.4.3 Application-related aspects

Once having created a basic Zebu specification consisting of the ABNF and associated protocol constraints specification, the developer can further annotate it according to application-specific requirements. Annotations define the message view available to the application, by indicating the message elements that this view should include. Other annotations control the degree to which the generated parser enforces the protocol constraints and the time when message element parsing occurs. Some of these annotations are illustrated in Figure 3.6. These annotations drive the generation of the data structures that contain the message elements, the stub functions and the parser.

Annotations on the right-hand side of a grammar rule select message elements that will be available to the application and impose type constraints on these elements. To make an element available, the programmer only has to annotate it with a colon and the name of a field in the associated data structure that should store the element's value. The field is then created by the Zebu compiler in the data structure containing the element. For instance, on line 3 of Figure 3.6, the Zebu programmer indicates that the application requires the method and the URI of the request line. Hence, the data structure representing the request line will

contain two string fields: `method` and `uri`. When there are no further annotations, the parser stores in these fields the starting position of the corresponding element within the message and its length. In some cases, however, it is also useful to control the type of the parsed value. This can be specified using the notation `as` followed by the name of the desired type. For example, in Figure 3.6 line 25, the `number` field of the CSeq header is specified to represent an unsigned integer of 32 bits (`uint32`). A type constraint not only enables representing an element as a type other than string, but also constrains its value by run-time assertions. The use of both kinds of annotations allows the generated data structures to be tailored to the requirements of the application logic rather than being strictly derived from the protocol syntax, as in other approaches [70, 69]. This reduces memory usage and simplifies the application logic's access to the message elements.

Annotations on the left-hand side of a grammar rule define the type to be used to represent the value generated from parsing a message according to the rule. The programmer annotates those non-terminals that should have specific base types or that should correspond to individual data structures. The `struct` annotation creates a C structure for the associated rule. This structure contains all the annotated fields obtained by derivation from this rule. Such an annotation can be seen in Figure 3.6, line 31. For each occurrence of `addr-spec` in the message, a dedicated data structure is created. The `enum` annotation, on the other hand, indicates that the only information required is the choice of alternative, and no information is required about the sub-components of these alternatives. Hence, the `Method` rule (line 16) is an enumeration of the possible methods. Annotating this rule as a enumeration enables efficient differentiation on the request based on the method name. Finally, the `union` annotation is provided for non-terminal alternatives. As for C unions, it combines several heterogeneous types into one type. Unions are implemented as C unions, with an additional tag identifying the union element.

The application logic does not directly manipulate the data structures declared by the Zebu specification and used in the protocol-handling layer. Instead, it accesses these data structures by stub functions generated by the compiler; examples are shown in Figure 3.7. Several kinds of stub functions exist: initializing, parsing, testing and data retrieving. The `init` function initializes data structure messages. `parse` is the parsing entry point and `parse_headers` triggers the header parsing. `isRequest` and `isResponse` are test stubs used to obtain higher-level information from the parsing such as the message kind. `get_RequestLine` is an example of a stub to retrieve data; it gets data representing the request line.

```
sip3261 init();
void parse(sip3261, char *, int);
void parse_headers(sip3261, E_Headers);
void parse_addr_spec(T_Lazy_addr_spec);
T_bool isRequest(sip3261);
T_bool isResponse(sip3261);
T_Str Option_Str_getVal(T_Option_Str);
T_RequestLine get_RequestLine(sip3261);
T_StatusLine get_StatusLine(sip3261);
T_header_From get_header_From(sip3261);
T_Method RequestLine_getMethod(T_RequestLine);
T_MethodEnum Method_getType(T_Method);
T_Str Method_getValue(T_Method);
T_Lazy_addr_spec header_From_getUri(T_header_From);
T_addr_spec Lazy_Addr_spec_getParsed(T_Lazy_addr_spec);
T_Option_Str Addr_spec_gethost(T_addr_spec);
```

Figure 3.7: Zebu compiler generated stubs

Inspired by the very efficient SER SIP parser [58], the parsing functions generated by the Zebu compiler implement a two-layer parsing strategy. A coarse-grained generic parser browses the message line by line until identifying a required high-level message element, such as the command line, a particular header or the message body. When the required high-level message element is identified, the generic parser pauses and a parser dedicated to this element starts its execution. Once this dedicated parsing completes, the coarse-grained parser may resume to continue the parsing, if more information about the message is needed. The dedicated parsers must respect both the ABNF specification and the constraints expressed in the Zebu annotations. The parsers store the message elements in the generated data structures while verifying the

constraints. The value of the named elements can then be accessed using the stub functions prefixed with *get*.

When the parsing of a message element is complex, and the information about this message element is only required in some situations, the parsing can be made incremental (or *lazy*). The Zebu annotation `lazy` specifies that a particular message element is parsed only when requested by the application. Examples are shown in Figure 3.6 with `Request-URI` and `addr-spec`. Incremental parsing avoids the parsing of a complex message element if the information it contains is not needed. Hence, the depth of nested structures in the message view is adjusted to application needs. Concretely, for an element that is to be incrementally parsed, the parser dedicated to the relevant command line or header only records in a data structure the starting position and the length of the element. The application can later call a stub function to initiate the actual parsing.

In addition to the stubs that control the parsing, the Zebu compiler also generates message modification stubs to be used by the application. These stubs are generated for each annotated element to update its value. However, the protocol specification may indicate that some message elements must not be modified. The `ReadOnly` keyword specifies this constraint. When an element is `ReadOnly`, no modification stub is generated, preventing any modification.

3.4.4 Specification verifications

Because a Zebu specification amounts to an ABNF specification from an RFC augmented with constraints and typing information, the Zebu compiler checks the consistency of the ABNF specification itself, and as well as the relationship between the ABNF and the constraints and type information.

Writing an ABNF specification can be difficult and there are currently no dedicated tools, such as editors, that verify the consistency of such a specification. While errors in the ABNF are unlikely in the RFCs of mature, widely used protocols, they may occur in specifications of new protocols or of protocol extensions. The Zebu compiler thus checks that the ABNF specification satisfies the following critical structural properties:

- **No Redefinitions** A specification rule can only be defined once. A rule can be extended using the '/=' operator, but must not be redefined using the '=' operator.
- **No Omissions** A rule that is referenced must be defined.

In addition, Zebu imposes the following constraint, due to its use of regular expressions in the parsing process.

- **No Cycle** Because the Zebu-generated parsing code is based on regular expressions, an ABNF specification must not contain any cycles, i.e., it should not be possible to derive a rule from itself after some successive derivations. Although support for cycles in the ABNF is required for exact parsing of some specific constructs, such as SIP comments, this limitation of the current implementation of the Zebu compiler does not prevent the processing of such information. Indeed, this kind of ABNF rule can typically be rewritten to eliminate the cycle at some loss of completeness, by encoding recursion up to a fixed depth, or at some loss of precision, by replacing a cycle by a repetition.

The Zebu compiler furthermore checks the following properties of the structure and type constraints.

- **Structure Constraints** Structure constraints expressed between curly brackets must be consistent. For instance, two elements constrained to be equal must present the same annotations to ensure the data types representing these elements are identical. Moreover, constraints specific to requests cannot refer to elements specific to responses, and vice versa.

- **Typing Constraints** The message view specification must be correctly typed with regards to the message grammar. More precisely, the type annotation must respect the potential values of the message elements. For instance, a non-terminal can be annotated as an integer only if it can be derived as a number.

3.5 The Zebu Toolchain

We now turn to the implementation of Zebu. We first give an overview of how to develop a network application using Zebu. We then present some of the optimizations implemented by the Zebu compiler, to allow it to generate code both for message parsing and message creation that is efficient and has low memory requirements. Finally, we consider the specific problem of integrating Zebu code into an embedded system, where the support for various operating system features essential for networking may be non-standard.

3.5.1 Developing a network application with Zebu

Fig. 3.8 illustrates the process of developing a network application using Zebu. The developer must first write the Zebu specification based on the protocol ABNF and the associated constraints. He can annotate the specification to indicate the parts of the message that the application needs to manipulate. He then gives the specification to the Zebu compiler, which first verifies it to detect potential inconsistencies and then generates the protocol-handling layer consisting of the data structures defined by the message view annotations and a parser to process messages and fill these data structures accordingly. This compiler amounts to 7,250 lines of OCaml code and 1,200 lines of C code, and targets C code. The protocol-handling layer also contains the compiler-generated stubs allowing the application to read and write message data. These stubs reflect the data structures specified by the provided annotations. The developer then defines the application as an *ordinary* C program and uses the stubs defined in the Zebu-generated protocol-handling layer to access the information related to the message content.

Fig. 3.9 illustrates the implementation of a Zebu-based application that extracts the caller host when the request is an `INVITE`. This kind of task is typically performed by a NIDS to track down particular message patterns. Initially, the application calls the stub functions `isRequest` to determine whether the message is a request and then `RequestLine_getMethod` and `Method_getType` to determine if the command line method is an `INVITE` (lines 5-6). The test compares the command line method to the `E_INVITEm` constant representing the `INVITEm` alternation in the `Method` rule (line 6). If the method is `INVITE`, the application then calls `parse_headers` to trigger the parsing of the `From` header (Fig. 3.9, line 8), which is incremental. The `get_header_From` and `header_From_getUri` stubs then extract the header URI (line 10).

Line 31 of the Zebu SIP specification (Fig. 3.6) indicates that parsing the URI must be incremental. Consequently, the `parse_addr_spec` function is used to actually parse it (line 11) and `Addr_spec_getParsed` is used to get the parsed value (line 13). After checking that the URI is defined (line 14), its value is extracted using `Option_Str_getVal` (line 16).

3.5.2 Message parsing

Zebu message parsing is based on regular expressions and is implemented using the Perl Compatible Regular Expressions (PCRE) [50] library. The parsing process is twofold. First, *message validation* checks the message format against the protocol grammar specification. Then, *message rendering* extracts some selected fragments and copies these fragments into data structures dedicated to the application. As HTTP-like messages can be complex, a straightforward implementation of these operations can be time-consuming and memory-intensive, which is often not acceptable for these very frequently performed operations. We thus present the optimizations we have implemented for these two steps.

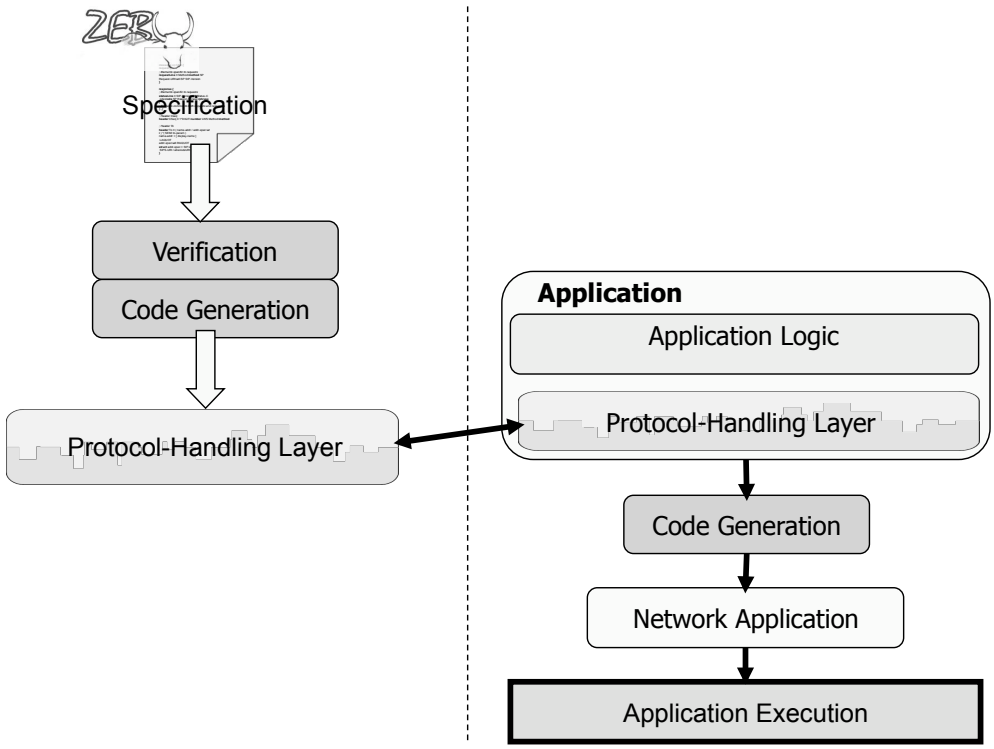


Figure 3.8: Zebu network application development process

Message Validation By default, the message validation code generated by the Zebu compiler strictly conforms to the protocol specification. The parser only accepts valid messages while rejecting all others. Hence, the robustness of the protocol-handling layer is guaranteed, but at the expense of performance and code size [21]. For instance, 50% of the code generated in the default case for the SIP protocol is devoted to message validation. For the arm9 architecture, this amounts to 86 kB of compiled code. However, if the application is used in a closed and secure environment, communicating only with trusted entities, this strict validation can be unnecessary. Thus, the developer can choose partial validation, in which case only annotated elements are validated. Moreover, the developer can annotate an element with `nocheck` if the application uses the element without requiring its validation or is able to postpone this validation.

When validation is not required, the message text still needs to be inspected to detect the beginning of the next annotated element. We have developed an algorithm to reduce the number of states in the part of the parsing automaton dedicated to non-validated message elements. This algorithm first identifies tokens that precede an annotated element but cannot be part of this element. These are considered as *bounds* that must be reached before the desired element. The states representing non-annotated elements except for these tokens are then merged, which can drastically reduce the automaton size.

As an example, if we have annotated `Request-URI` with `nocheck` (Fig. 3.6), the parser must only find the beginning and end of the URI value. Since the `Request-URI` subfield is followed by a space character and since a space cannot occur within a URI, several states of the parsing automaton dedicated to the URI are merged to collect only non-space characters. Fig. 3.10a illustrates a fragment of the regular expression generated by the Zebu compiler from the automaton with full validation for `Request-URI`. The complete regular expression represents 12784 characters. The complete regular expression after reduction is shown Fig. 3.10b and illustrates the reduction factor obtained.

```
1 sip3261 msg = sip3261_init();
2 sip3261 msg = sip3261_parse(msg, buf, len);
3 if (isRequest(msg)) {  // Process only request messages
4  T_RequestLine requestLine = get_RequestLine(msg);
5  T_Method method = RequestLine_getMethod(requestLine);
6  if (Method_getType(method) == E_INVITEm) {
7    // Filter INVITE methods
8    parse_headers(msg, E_HEADER_FROM);
9    T_Lazy_addr_spec l_addr_spec =
10   header_From_getUri(get_header_From(msg));
11   parse_addr_spec(l_addr_spec);
12   T_Option_Str host =
13     Addr_spec_getParsed(l_addr_spec);
14   if (Option_Str_isDefined(host)) {
15     // host may be undefined in some cases, check it and log its value
16     mylog(Option_Str_getVal(host));
17 }}}
```

Figure 3.9: Excerpt of a SIP statistics application based on Zebu

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

3.10a. Strict validation (excerpt)

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

3.10b. Permissive validation (complete)

Figure 3.10: Generated regular expressions

Message Rendering A Zebu specification contains annotations defining the data structures representing messages. The stubs generated by the Zebu compiler copy elements from the message into local data structures. However, the application often uses the raw value of an element, as a string. In this case, copying into intermediate data structures can be avoided. Indeed, the only difference between the raw element value in the message and the value required by the application is a `'\0'` character at the end. If a message element is not immediately followed (without a separator) by another requested element, the stub can swap the character immediately after this message element with a `'\0'`. Instead of pointing to a fresh buffer filled with the element value, the data structure keeps a reference towards the actual position of the element in the message.

This strategy, however, corrupts the original message, which may be needed to retransmit the message or create a new one. To be able to reconstruct the original message, the stub also stores the swapped character. This technique is known as *character-swapping* and is notably implemented in Lex [63]. Using this technique, no memory allocation and no copying are required.

3.5.3 Message creation

Usually, a protocol support library provides the developer with a set of functions to create, initialize and modify message fragments [41]. Because the size of the message fragments resulting from the message modifications cannot be determined statically, the provided functions typically operate on a multi-layered memory representation, with nested structures pointing to each message element. Once all the modifications have been performed, the application translates these data structures into a textual message. The generated message is then stored in a fresh buffer that can be sent when requested by the application.

However, this approach requires many memory copies and many intermediate data structures due to the multi-layering.

The code generated by the Zebu compiler does not create a multi-layered data structure that reflects the message structure, but instead maintains a flat string representation and only collects the information related to the message modifications. Concretely, the Zebu compiler provides functions to create a new message or to copy an existing one, and functions to update the various elements of a message. These updating functions do not actually modify the message, but instead store modifications as *lumps*. A *lump* is a data structure that describes the modification to perform: the modification, where it must be applied and what is its effect on the message size. All the lumps are stored in a linked list sorted according to the modification position. Several kinds of lumps are provided: adding, updating and removing a complete header and adding, updating and removing a message element. The use of lumps minimizes the creation and allocation of intermediate data structures to represent the effects of the modifications.

Lumps are applied to a message only when it is about to be sent. At this time, a new buffer is allocated; its size corresponds to the original message size offset by the effect of the accumulated modifications. The beginning of the original message is copied into this new buffer until reaching the position of the first lump. The modification described by the lump is then applied. This process iterates over the whole lump list until the end of the new message. Finally, the application can store this buffer to update it later, if the same message or a sufficiently close one must later be sent.

3.5.4 Generating an execution environment for an embedded system

The code generated by the Zebu compiler cooperates with an underlying run-time support that provides low-level functionalities such as *sockets*, *multi-threading* and *memory allocation*. The support available for these functionalities depends on the operating system. As operating systems for embedded systems typically provide fewer functionalities than general-purpose operating systems, integrating a Zebu-based application into an embedded system can be cumbersome. Moreover, some embedded operating systems, such as FreeRTOS, require the application to be part of the operating system implementation.

To overcome these problems, we have developed a generator of run-time support, ZebuRTGen, depicted in Fig. 3.11, which generates a runtime system for a particular embedded system according to the requirements of the application code and the code generated by the Zebu compiler. So far, we have implemented two back-ends for ZebuRTGen, one for Linux and one for FreeRTOS. These back-ends consist of generic components that can be customized according to the protocol specification.

One example of the included generic components and their customization is a socket server with TCP and UDP support. In the case of a session-oriented protocol, such as TCP, the end of a message occurs when the transport session ends. However, such information is not available with a datagram-oriented transport protocol such as UDP. Consequently, the socket server must process received messages to determine if the end of message has been reached. In this case, the protocol specification defines how to detect the message end. For instance, the SIP protocol relies on the `Content-Length` header to specify the body length. ZebuRTGen customizes the socket server with the code generated by the Zebu compiler to detect this header and deduce the total message size.

3.6 Robustness

Because message processing code represents the interface between a network application and the outside world, it must be robust. Indeed, the consequences of faulty protocol-support can be dramatic, ranging from crashing the application to providing an opportunity for an attacker to remotely take control of the platform. Several security alerts show that errors in protocol support can survive for a long time without being detected. For instance, through four successive versions of the Helix Streaming Server, errors in the

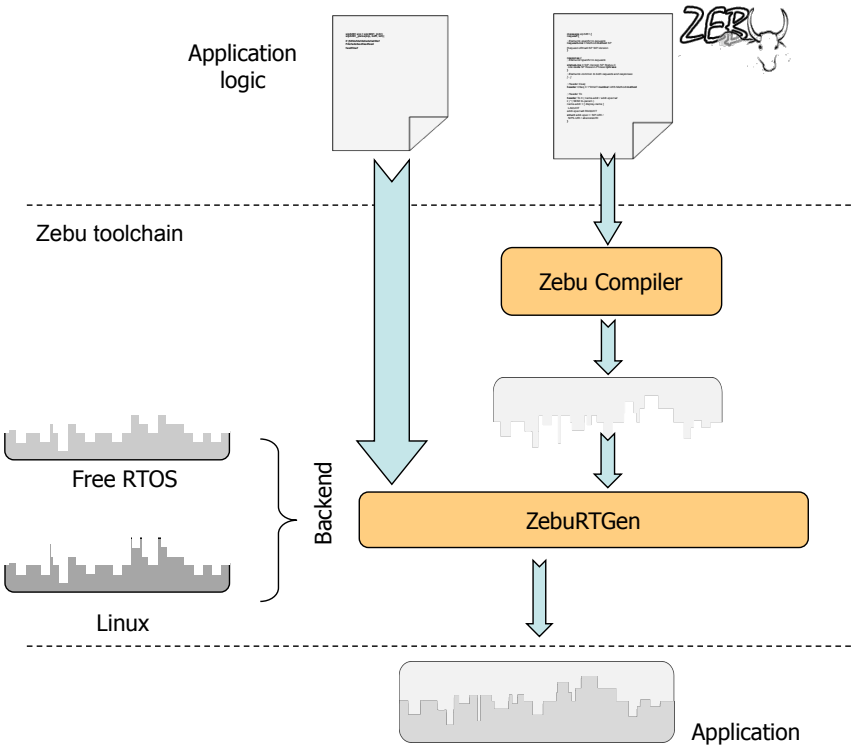


Figure 3.11: ZebuRTGen usage in the Zebu toolchain

parsing of a URI enabled an attacker to execute arbitrary code in the server.[4]

In this section, we assess the impact of using Zebu on the robustness of the protocol support of network applications. We first present the analysis technique, inspired by mutation analysis [36], used to evaluate the robustness of message parsers. This technique measures how well a parser covers a specification by sending it various messages, which may or may not be valid. Messages are constructed according to a set of mutation operators derived from message patterns extracted from an RFC that lists potentially problematic messages [91].

3.6.1 Mutation analysis

The concept of mutation analysis [36] was introduced at the end of the 1970s to test program viability. The main principle of this approach is to mutate a correct program by injecting a fault into it. Mutations are not applied randomly but follow a set of mutation rules. These rules are generally deduced from frequent programming errors [2].

A mutation analysis is parameterized by a fault class. A fault class provides a set of operators, called mutation operators, that model one or more faults in the identified fault class. A single mutation operator is applied to a single entity in the program to be tested, transforming the program into an almost identical faulty program, known as a mutant. Once generated, a mutant is applied to a test suite. If, for any test, the result of the mutant differs from the result of the original program, the mutant is said to be killed. The percentage of dead mutants defines how well the program is covered by the test suite.

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

In our study, we measure the coverage of the properties verified during the message validation phase. To provide uninterrupted service, a robust network application must accept all messages that are valid according to the specification. A robust network application must also reject (or at least detect) all invalid messages to avoid corrupting its internal state. Our technique differs from the original mutation analysis in how mutation operators are applied. In our case, the operators are not applied to the program but to the input messages to validate the program's parsing behavior. A set of original valid messages is modified according to mutation rules to identify the consequences that ensue from the introduction of an error.

3.6.2 Mutation rules

In our study, the protocol support of a network application is subjected to a set of deliberately pernicious messages to validate its behavior. We have based our analysis on a set of mutation rules derived from RFC 4475 [91], which defines a test suite for use in certifying implementations of the SIP protocol. We have focused on messages or patterns targeting parsers. The RFC 4475 identifies a dozen message kinds that are valid but potentially problematic and about twenty invalid messages. Many of these messages, however, can be generated by the application of the same mutation operator. We have thus identified four mutation operators that render a message invalid and three that transform it into a pernicious but valid one. These mutation operators are specific to HTTP-like protocols.

Mutation operators producing invalid messages

Based on the RFC 4475, we have defined a set of mutation operators attacking characters, character sequences, integer values and literals, as specified in the protocol grammars. Applying these operators results in mutants embodying a wide range of errors.

Character Mutations A message element is a sequence of characters picked from a corresponding character set. The mutation operator inserts a character that is not in this character set. A mutated character is inserted at the beginning or at the end of a message element, or at an arbitrary position within the message element. Thus, for each character outside of the message element's character set, three mutants are produced.

Sequence Mutations A message element is defined by both the character set and the number of repetitions allowed for the character set. The number of repetitions defines indirectly the minimal and maximal length of a string. For instance, the response code in HTTP must be three digits long; otherwise, the message is invalid. We thus define a mutation operator that uses valid characters but an invalid number of repetitions. One mutant is generated with a number of characters below the lower bound and another with a number of characters above the upper bound.

Integer Mutations We distinguish the case when the character set is exclusively made of digits, because many applications parse numbers differently than words. One mutation operator inserts alphabetical characters into numbers. Another mutation operator creates a string that represents a value that is outside the associated bounds. Such bounds are usually informally defined in the text accompanying the specification.

Literal Mutations An element is called a literal when it can take the form of any string found in an enumeration. The mutation operator derives mutants by merging a string of this enumeration with an arbitrary character. This mutation tests whether the parser simply looks for a particular pattern within the string to be parsed.

Mutation operators producing valid messages

Producing valid messages that can potentially corrupt parsers can be difficult. Actually, most of the parsers tested process most valid messages in an appropriate way. Thus, we focus on some subtleties that appear in specifications and that have been found to cause errors in the considered protocol support.

Protected Characters In many cases, some special characters can appear within a message element when preceded by some protective character. The implementation of this protection is often defective. To test this feature, we introduce protected special characters within message elements, the most basic example being a protected space, *i.e.* a back-slash followed by a space.

Element Length Many errors stem from the bad handling of array lengths and more generally of buffers. Indeed, the arrays used to store the various elements extracted from a message are often statically allocated with a small size, introducing the possibility of buffer overflows. We introduce a mutation operator that checks that the parser respects the constraints on element length. This operator pads elements to verify that the parser can handle long elements that are within the maximal length defined in the specification.

Extensions Some message elements are defined according to a set of alternatives that allow for extensions. The mutation operator creates values respecting the extension clause that are close to one of the enumerated alternatives. For instance, one mutant uses an existing alternative as a prefix. Indeed, in some parsers that we have studied, as soon as an alternative is detected as a prefix, the parser considers it to be complete even if some trailing characters remain. The trailing characters are then considered to be part of another message element.

3.6.3 Experiments

Our objective is to assess the impact of using Zebu on the robustness of parsers for HTTP-like protocols. We evaluate this robustness by comparing the behavior of Zebu-based parsers to existing SIP and RTSP parsers. We stress the parsers with invalid messages and with valid but potentially problematic messages.

We compare the SIP Zebu-based parser with the parser provided by the Osip library [75] and the one found in the SER proxy server [58]. We compare the RTSP Zebu-based parser with the parser provided by the LiveMedia library [65] and the one found in the multimedia streaming server VLC [96]. Figure 3.12 summarizes the sizes of the various considered parsers, all written in C, as well as the sizes of the original ABNF specifications and Zebu specifications. The SIP Zebu specification contains twelve times fewer lines of code than the SER parser, which is recognized as a reference in the SIP community. In the case of RTSP, the ratio is smaller, mainly because the considered parsers are very simple. This table shows also that the Zebu specifications are around 50% larger than corresponding ABNF specifications. This is in part because ABNF specifications are potentially incomplete since they may refer to other ABNF specifications (*e.g.* the URLs in HTTP), and in part because the structural constraints integrated in a Zebu specification can require a significant number of added lines.

Protocol	ABNF Size	Zebu Specification Size	Parser	Parser Size
SIP	≈ 700	1081	oSIP	11982
			SER	13277
RTSP	≈ 200	330	VLC	≈ 1200
			LiveMedia	≈ 1000

Figure 3.12: Line numbers for the grammars and variousparsers of SIP and RTSP

Experimental Methodology

To assess the robustness of the generated parsers, we have implemented several versions of a small application that focuses on the command line and mandatory headers of SIP and RSTP messages. This application logs some particular message elements for statistical purposes. The Zebu-based applications, `Zebu-SIP` and `Zebu-RTSP`, consist of a few lines of C code calling the stub functions generated by the Zebu compiler (Figure 3.7). The Zebu-generated parsers used for these experiments conduct a strict analysis of a message with total validation of its elements.

We have implemented the same logging application in a SER proxy. SER provides two levels of programmability. A first level is introduced through a dedicated configuration language. This language provides high level constructs to access the various elements of a message; these constructs rely on low-level functions implemented in the core of the server. The second level is available by leveraging a module system to create functions with direct access to messages. These functions can then be exported to the configuration language. We have chosen to implement the `SER` version using the configuration language to benefit from the existing parsing functions. The other versions, `oSIP`, `VLC` and `LiveMedia`, are written in C using the various functions provided by their integrated parsers.

In each version, we have instrumented the protocol support to record its behavior when receiving a message. Notably, we monitor the various error flags that may be raised when a message is detected to be invalid.

Invalid Messages

As shown in Figure 3.13, the manually developed parsers considered do not detect more than 28% of the invalid messages. This gap stems in part from a certain flexibility of these parsers with respect to the specification. Flexibility is indeed generally recommended in protocol specifications, which argue that *borderline* messages should be accepted to avoid isolating applications that are not 100% compliant with the specification. However, the support for flexibility is often not very well controlled and consequently some invalid message classes are wrongly handled. The parser of the Videolan streaming server is extremely lax in its parsing. Only 0.1% of the transmitted messages were detected as invalid by its protocol support. Examining the code shows that it is very simplistic. On the other hand, the strict Zebu-generated parsers detect 100% of the invalid mutated messages of our test suite. This result is made possible by generating complete and strict parsers.

		Invalid Messages	Detected Messages	% Detected Messages
SIP	`oSIP`	5976	1020	**17.1%**
	`SER`		1512	**25.3%**
	`Zebu-SIP`		5976	**100.0%**
RTSP	`VLC`	2730	4	**0.1%**
	`LiveMedia`		748	**27.4%**
	`Zebu-RTSP`		2730	**100.0%**

Figure 3.13: Parser coverage for invalid SIP and RTSPmessages

Our experiments also triggered a bug in the SER proxy server. This server keeps track of all the erroneous `Via` headers. When receiving a large number of invalid messages, this collection failed, entailing a segmentation fault. This error, now corrected, was present during a year (from version 0.8.14 of July 2004 to version 0.9.3 of September 2005).

Valid Messages

Figure 3.14 shows that the manually-developed parsers reject some valid messages. These messages are of course unusual but rejecting valid messages demonstrates that the flexibility introduced in these parsers is not really controlled. Worse, some valid requests crash the oSIP library parser. The SIP Zebu-generated

parser rejects no valid message. Hence, the Zebu-generated parser detects all invalid messages while rejecting no valid messages of our test suite.

		Valid Messages	Rejected Messages	% Rejected Messages
SIP	Zebu-SIP	549	0	**0.0%**
	oSIP		21	**3.9%**
	SER		2	**0.4%**

Figure 3.14: Parser coverage for valid SIP messages

We have run the same experiments for the RTSP protocol. None of the considered parsers rejects a message. This is not surprising because the manually-developed parsers are very lax.

3.7 Performance

All of the messages received by and sent from a network application go through the protocol-handling layer. Consequently, the performance of the protocol-handling layer greatly affects the performance of the application. In this section, we assess the performance overhead induced when using Zebu, by comparing the performance of Zebu generated code with the performance of manually-developed code. We compare our approach to the *de facto* standard for each considered protocol. Then, we evaluate the memory consumption of the code generated by the Zebu toolchain. This evaluation should be considered as a study of the feasibility of deploying Zebu-based network applications on embedded systems in real conditions and consequently the feasibility of using HTTP-like protocols in this context.

3.7.1 Processing time

We first present a micro-benchmark on HTTP, and then exhibit the results for the SIP-based application presented in the previous section. In both cases, these results are obtained with real messages. In our experiments, a client application replays a real trace, extracting and sending each message of this trace as fast as possible to stress a server application. We have instrumented the code of the various applications to measure the parsing time for each received message. For these experiments, the client is hosted by an Athlon Mobile (1.6GHz) and the server by a Pentium 4 (2.66GHz).

HTTP Micro-Benchmarks

In order to evaluate the processing time of one message by the Zebu-generated HTTP support, we developed an application that records the domain name contained in the `HOST` header. The HTTP Zebu specification amounts to 500 lines and entails the generation of 2400 lines of C code. The application consists of only 27 lines of C code. We have compared our Zebu implementation to an implementation based on PADS and to an implementation based on the Apache Web server.

The application based on PADS relies on the code generated from the HTTP PADS specification provided with the PADS compiler. This specification (approximately 2000 lines) consists of an ordered set of PADS type rules and regular expressions. The application represents 50 lines of C code, most of which is devoted to initializing the runtime environment.

The application based on the Apache web server is implemented in the web server's configuration file. This file enables fine tuning the behavior of the server through functions exported by the module system. Each module is dedicated to a particular feature, such as XML support or authentication. The module we are interested in is `mod_setenvif`, which enables defining parsing functions based on regular expressions for each message header. To implement our application we have added 9 lines to the original configuration file. These lines filter messages to record only those that contain a `HOST` header and extract the requested information.

The Zebu-generated parser performing complete message validation requires 170,000 cycles on average to parse a message while the PADS-generated parser requires 2,600,000 cycles on average. The parsing in the Apache server requires only 24,000 cycles but it concentrates only on the `HOST` header; the other headers are only referenced by a pointer without any further analysis.

		`SER-module`		`SER`		`oSIP`		`Zebu-SIP`		`Zebu-SIP-minimal`	
		Cycles	**Ratio**	**Cycles**	**Ratio**	**Cycles**	**Ratio**	**Cycles**	**Ratio**	**Cycles**	**Ratio**
Request	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%
Response		35 608	151%	35 716	152%	505 312	2 150%	31 806	135%	23 508	100%

Figure 3.15: Performance of the logging application on a trace of 3000 real SIP Messages (ratio with `Zebu-minimal`)

SIP Evaluation

In order to evaluate the performance in a SIP context, we have re-used the application described in Section 3.6.3. We have collected the SIP traffic exchanged at the University of Bordeaux during one day, amounting to 3000 messages. The Zebu-based implementations, `Zebu-SIP` and `Zebu-SIP-minimal`, each represent 27 lines of C code relying on two protocol-handling layers that differ in their strictness. `Zebu-SIP` performs complete validation of each received message as in Section 3.6.3, while the new version `Zebu-SIP-minimal` validates only the annotated message elements. To assess our results, we have reused the versions of the application based on SER and oSIP described in Section 3.6.3.

`SER` is implemented in SER by 12 lines of the SER configuration language, leveraging the ability to switch to an arbitrary `shell` script to extract the information from a message using a regular expression. This time, we have also implemented `SER-module`, a version relying on a dedicated module implemented using 150 lines of C code. The version based on oSIP consists of 40 lines of C code. The results of this evaluation are summarized in Figure 3.15 and compared to the performance of `Zebu-SIP-minimal`.

The parsing performed by `SER-module` is particularly efficient, up to 3 times faster than when using `Zebu-SIP-minimal` for requests. For responses, the parsing performed by `SER-module` is 50% slower than when using `Zebu-SIP-minimal`. The main reason is that SER is dedicated to message routing and thus always processes some message elements, such as the *Via* header, that are crucial for routing but are useless in our context. As soon as it recognizes a response, the Zebu-generated parser does not process the rest of the message. Zebu can thus provide better performance by generating a parser dedicated to a particular application, while guaranteeing its robustness, which can be compromised using SER modules. SER modules have full access to the internal data structures of the server which is a major threat to the server when these modules are erroneous.

The `SER` version is 217 times slower for `INVITE` messages mainly because of the context switch induced by the shell script call. This call is only performed when an `INVITE` message is identified.

The parsing performed by `oSIP` is between 5 and 10 times slower than `Zebu-SIP-minimal` for requests and around 20 times slower for responses. In both cases, the oSIP library parses the six mandatory headers and stores for each header a pointer to its starting position.

3.7.2 Memory usage

The previous experiments have shown that a Zebu-based approach is a good alternative to using existing frameworks in terms of both robustness and performance. In this section, we evaluate the memory requirements of the Zebu-based approach, and assess whether the generated parsers can be used in embedded systems.

Our Experiments

Our experiments focus on a building surveillance application deployed in our test environment. This application enables a SIP phone to control and receive images from an IP camera. The camera only supports the RTSP protocol for negotiating multimedia parameters and the HTTP protocol for operating commands such as moving or zooming. New code cannot be loaded into the camera. Thus, to allow a SIP phone to interact with the camera, we had to create a gateway to convert SIP messages into RTSP or HTTP, according to the requested command. This gateway involves both parsing incoming messages and constructing SIP, RTSP and HTTP messages. Our experiments focus on the gateway implementation as an embedded system, as would be necessary in a real building automation context.

We have deployed our gateway on two kinds of embedded systems: an Eukrea CPUAT91 board with an ARM9 and 32 MB of SDRAM with Linux 2.6.20 installed, and an ATMEL EVK1100 development kit based on an AVR32 micro-controller with 32 MB of RAM with FreeRTOS installed. The SIP client used to control the camera is a slightly modified version of linphone [64].

Figure 3.16 illustrates the various components involved in our application. A SIP client sends an `INVITE` message to the gateway to request video streaming from the camera. The gateway extracts some information, such as the caller identifier, from the SIP message and integrates it into a newly-crafted RTSP message that is then sent to the camera. Once the communication is established, the gateway is no longer concerned, and the multimedia flow is transmitted directly from the camera to the SIP client via the RTP protocol.

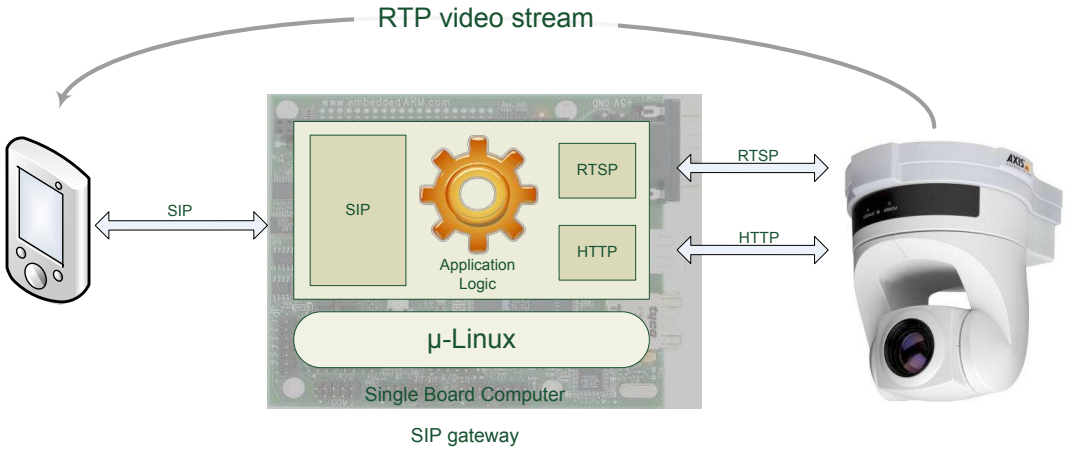


Figure 3.16: SIP Gateway for the Axis Camera

To control the various functionalities provided by the camera, such as zooming and moving, we have extended the SIP protocol with new methods and headers representing these features. For instance, in order to command the camera to zoom, we introduce a new SIP method `ZOOM` and a new header `Zone` that describes which part of the captured scene should be in focus and the zoom factor. Figure 3.17 shows the lines that must be added to the Zebu specification to support this extension in the gateway. At the same time, we have extended the SIP client to support these extensions. Thus, a SIP phone running this extended client can receive images from the camera and control its movement. When the gateway receives a SIP message defined according to a new extension, it extracts some information to build the corresponding HTTP request and sends it to the camera. Conversely, it converts the HTTP response returned by the camera into a SIP response.

We have implemented four versions of the gateway on the CPUAT91 board: two based on Zebu, one based on the eXosip library [41] and one based on the Sofia-SIP library [87]. We have also implemented two versions using the EVK1100 development kit using our FreeRTOS back-end. One Zebu based version performs full validation while the other only validates annotated elements.

```
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
```

Figure 3.17: Zebu Specification of the Zoom Extension

Static Memory Requirements

First, we consider the static memory requirements of the gateway compiled for the ARM9 based board. These memory requirements comprise those of the Linux kernel, the protocol-handling layer, and the application itself. The Linux kernel using a minimal root filesystem requires 1.8 MB. Figure 3.18 shows the size of the protocol-handling layer for each version. With full validation, the Zebu generated code uses only 35% of the memory needed by eXosip and 11% of what is needed by Sofia-SIP. With no validation, the gap is wider, with the Zebu generated code using only 13% of what is needed by eXosip and 4% of what is needed for Sofia-SIP.

Zebu		eXosip	Sofia-SIP
Complete Validation	No Validation		
139	53	400	1 300

Figure 3.18: Size, in kB, of the protocol support compiled for ARM9

We now consider the static memory requirements of the gateway when compiled for the AVR32-based EVK1100 board based on an AVR32. For our experiments, we have used ZebuRTGen with the FreeRTOS back-end to generate a complete application integrated in the operating system. Figure 3.19 shows that the ROM requirements are contained within the 512 kB available on the board. The static RAM requirements amount to only a tiny part of the 32 MB available.

	ROM	RAM	Total
Complete Validation	234.7	64	298.7
No Validation	150.6	64	214.6

Figure 3.19: Memory usage for the complete gateway on the Eukrea board (in kB)

Dynamic Memory Requirements

In order to evaluate the dynamic memory requirements of our gateway, we consider a set of fixed actions. First, the client sends an `INVITE` request to the gateway. Once the connection is established, the client sends a sequence of `ZOOM` requests to modify the zoom factor and the considered zone, and then closes the connection. In order to measure the memory consumption, we have used the `exmap-console` tool [40].

Figure 3.20 shows the maximum amount of memory used to execute the `INVITE`/`ZOOM`/`BYE` sequence for each version of the gateway. The memory consumption only depends on the message currently processed, as no state is kept in the gateway. The *Virtual Memory* column indicates the address space allocated for the gateway, while the *Physical Memory* indicates the amount of physical memory actually used by the gateway. The implementations based on Zebu require only 40% of the virtual memory required by manually-developed protocol support. Furthermore, complete validation only adds 15% of required physical memory. Zebu implementations consume between 51% and 59% of the memory consumed by eXosip and 35% and 41% of the memory consumed by Sofia-SIP.

We furthermore observe that the memory reduction does not affect the performance. Figure 3.21 shows the parsing time for an `INVITE` SIP request initiating the communication with the gateway. This message

		Virtual Memory	Physical Memory
Zebu	Complete Validation	1 972	632
	No Validation	1 972	548
eXosip		4 488	1 068
Sofia-SIP		5 056	1 544

Figure 3.20: Process memory footprint in kB

represents the worst case for parsing because it includes several complex elements required to establish the communication. With complete validation, the parsing takes around 3500 μ-seconds, which is two times slower than Sofia-SIP. However, without validation, the Zebu parsing competes with Sofia-SIP and is twice as fast as eXosip.

Zebu		eXosip	Sofia-SIP
Complete Validation	No Validation		
3 513	1 517	3 849	1 489

Figure 3.21: Parsing time of a SIP Message in the worst case (in μ-seconds, average of 10 tests)

Finally, we analyze the physical memory and virtual memory consumption of each layer of the gateway. The gateway can be divided into three distinct layers: (1) the runtime environment, which comprises the commodity functions used by the application, (2) the protocol-handling layer and (3) the application. The runtime environment and protocol-handling layer are specific to each implementation. Figure 3.22 shows the peak memory consumption for each of the implementations.

If we focus on the protocol-handling layer, Sofia-SIP is the fastest but consumes the most memory with 1.1 MB of virtual memory and 640 kB of physical memory. This represents 85% of the total memory consumption for the Sofia-SIP gateway. The Sofia-SIP parser creates many data structures. Each header is copied into a fresh buffer, then parsed and the parser stores each element in a fresh data structure. Its message modification strategy is also flexible but has high memory usage.

	Application based on Zebu (complete validation)		Application based on Sofia-SIP				Application based on eXosip			
	Memory		**Memory**		**Ratio**		**Memory**		**Ratio**	
	Virt.	**Phys.**	**Virt.**	**Phys.**	**Virt.**	**Phys.**	**Virt.**	**Phys.**	**Virt.**	**Phys.**
Runtime Environment	96	60	160	92	167 %	153 %	260	124	271 %	207 %
Protocol Support	188	44	1116	640	594 %	1 455 %	404	360	215 %	818 %
Application	16	16	16	16	100 %	100 %	16	16	100 %	100 %

Figure 3.22: Memory consumption per layer (in kilobytes, ratio as compared to the Zebu-based application)

The protocol handling layer of the eXosip-based implementation requires far less memory but still consumes 72% of the total memory consumption of the gateway. EXosip is a wrapper around oSIP providing higher-level constructs. Notably, eXosip conceals the SIP transactional model with dedicated data structures. However, eXosip reuses the parsing phase of oSIP which is quite similar to that of Sofia-SIP. The only difference is in the message creation and modification which avoid useless copies. A Zebu based implementation with complete validation requires less than the half of the virtual memory required by eXosip and less than the eighth of the physical memory.

3.8 Conclusion

In this chapter, we have presented the Zebu domain-specific language, which has the goal of easing the development of network application protocol-handling layers while guaranteeing the robustness and performance properties. The Zebu language is based on the ABNF formalism used for protocols specifications

and enables fine tuning the protocol-handling layer according to the needs of a particular application. The programming effort is reduced by staying close to the original protocol specification, consequently reducing the opportunity for errors. Moreover, Zebu allows extending, in an easy and safe way, a protocol underlying an application to integrate particular needs. The various experiments presented in this chapter have demonstrated that this approach is a reliable alternative to manual development in the context of protocol-handling layers. Zebu generated code has good performance and has a significantly lower memory footprint than comparable existing manually encoded solutions.

Chapter 4

Z2z: Automatic Generation of Network Protocol Gateways

This chapter presents z2z [15, 7], a generative approach to gateway construction based on a domain-specific language for describing protocol behaviors, message structures, and the gateway logic. Z2z includes a compiler that checks essential correctness properties and produces efficient code. We have used z2z to develop a number of gateways, including SIP to RTSP, SLP to UPnP, and SMTP to SMTP via HTTP. Our evaluation of these gateways shows that z2z enables communication between incompatible devices without increasing the overall resource usage or response time.

4.1 Introduction

The *home of tomorrow* is almost here, with a plethora of networked devices embedded in appliances, such as telephones, televisions, thermostats, and lamps, making it possible to develop applications that control many basic household functions. Unfortunately, however, the different functionalities of these various appliances, as well as market factors, mean that the code embedded in these devices communicates via a multitude of incompatible protocols: SIP for telephones, RTSP for televisions, X2D for thermostats, and X10 for lamps. This range of protocols drastically limits interoperability, and thus the practical benefit of home automation.

To provide interoperability, one solution would be to modify the code, to take new protocols into account. However, the code in devices is often proprietary, preventing any modification of the processing of protocol messages. Even if the source code is available, it may not be possible to install a new implementation into the device. Therefore, gateways have been used to translate between the various kinds of protocols that are used in existing appliances.

Developing a gateway, however, is challenging, requiring not only knowledge of the protocols involved, but also a substantial understanding of low-level network programming. Furthermore, there can be significant mismatches between the expressiveness of various protocols: some are binary while others are text-based, some send messages in unicast while other use multicast, some are synchronous while others are asynchronous, and a single request in one protocol may correspond to a series of requests and responses in another. Mixing this complex translation logic, which may for example involve hand coding of callback functions or continuations in the case of asynchronous responses, with equally complex networking code makes implementing a gateway by hand laborious and error prone. Enterprise Service Buses [25] have been proposed to reduce this burden by making it possible to translate messages to and from a single fixed intermediary protocol. Nevertheless, the translation logic must still be implemented by hand. Because each pair of protocols may exhibit widely differing properties, the gateway code is often not easily reusable.

This chapter

We propose a generative language-based approach, z2z, to simplify gateway construction. Z2z is supported by a runtime system that hides low-level details from the gateway programmer, and a compiler that checks essential correctness properties and produces efficient code. Our contributions are:

- We propose a new approach to gateway development. Our approach relies on the use of a domain-specific language (DSL) for describing protocol behaviors, message structures, and the gateway logic.
- The DSL relies on advanced compilation strategies to hide complex issues from the gateway developer such as asynchronous message responses and the management of dynamically-allocated memory, while remaining in a low-overhead C-based framework.
- We have implemented a compiler that checks essential correctness properties and automatically produces an efficient implementation of a gateway.
- We have implemented a runtime system that addresses a range of protocol requirements, such as unicast vs. multicast transmission, association of responses to previous requests, and management of sessions.
- We show the applicability of z2z by using it to automatically generate a number of gateways: between SIP and RTSP, between SLP and UPnP, and between SMTP and SMTP via HTTP. On a 200 MHz ARM9 processor, the generated gateways have a runtime memory footprint of less than 260KB, with essentially no runtime overhead as compared to native service access.

Outline

The rest of this chapter is organized as follows. Section 4.2 presents the range of issues that arise in implementing a gateway, as illustrated by a variety of case studies. Section 4.3 describes the z2z gateway architecture and introduces a DSL for describing protocol behaviors, message structures, and the gateway logic. Section 4.4 describes the dynamic semantics of z2z and its interaction with the runtime system. Section 4.5 describes the compiler and runtime system that support this language. Section 4.6 describes the model checker provided by z2z for verifying protocol correctness. Section 4.7 demonstrates the efficiency and scalability of z2z gateways. Section 4.8 discusses related work. Finally, Section 4.9 concludes the chapter with some final remarks.

4.2 Issues in Developing Gateways

A gateway must take into account the different degrees of expressiveness of the source and target protocols and the range of communication methods that they use. Each of these issues requires substantial expertise in network programming, and the need to address both of them at once makes gateway development especially difficult. We illustrate these points using examples that involve a wide range of protocols.

Mismatched protocol expressiveness. The types of messages provided by a protocol are determined by the kinds of exchanges that are relevant to the targeted application domain. Thus, different protocols may provide message types that express information at different granularities. To account for such mismatches, a gateway must potentially translate a single request from the source device into multiple requests for the target device, or save information in a response from the target device for use in constructing multiple responses for the source device.

The SIP/RTSP gateway shown in Figure 4.1 illustrates the case where the requests accepted by the target device are finer grained than the requests generated by the source device. This gateway has been used in

the SIP-based building-automation test infrastructure at the University of Bordeaux. It allows a SIP based telephony client to be used to receive images from an Axis IP-camera.[1] This camera is a closed system that accepts only RTSP for negotiating the parameters of the video session. Once the communication is established, the gateway is no longer involved, and the video is streamed directly from the camera to the SIP client using RTP [82]. Because SIP and RTSP were introduced for different application domains, there are significant differences in the means they provide for establishing a connection. Thus, as shown in Figure 4.1, for a single SIP `INVITE` message, the gateway must extract and rearrange the information available into multiple RTSP messages.

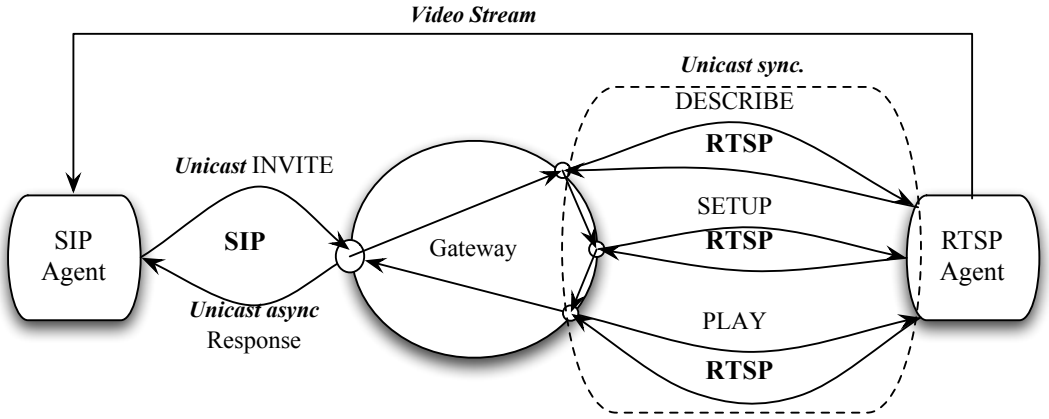


Figure 4.1: SIP to RTSP gateway

The SMTP/HTTP and HTTP/STMP gateways shown in Figure 4.2 illustrate the case where information must be saved from a target response for use in constructing multiple responses for the source device. These gateways are used in a tunneling application that enables SMTP messages to be exchanged between two end-points over HTTP, as is useful when the port used by SMTP is closed somewhere between the source and the destination. The first gateway encapsulates an SMTP request into an HTTP message and sends it asynchronously using UDP to the second gateway, which extracts relevant information to generate the corresponding SMTP request. The response is sent back similarly.

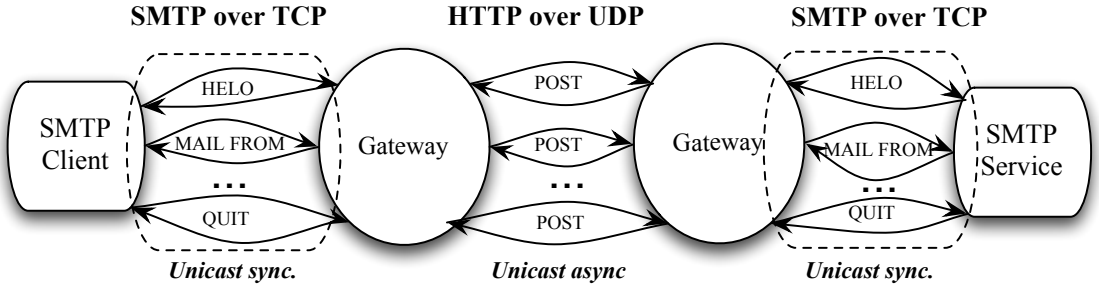


Figure 4.2: HTTP tunneling gateways

Because at the clien end and at the service end all SMTP messages have to flow within the same TCP stream, the HTTP/SMTP gateway needs to know which TCP connection to reuse when an HTTP request is received. To address this issue, the gateway generates a unique identifier when opening the TCP connection with the destination SMTP server and includes this identifier within the HTTP response. The first gateway then includes this identifier in all subsequent HTTP requests, enabling the second gateway to retrieve the connection to use. To implement this, the first gateway needs to manage a state within a session to store the identifier returned in the first response in order to be able to find it for the subsequent requests.

[1] Axis: http://www.axis.com/products/

Heterogeneous communication methods. Protocols differ significantly in how they interact with the network. Requests may be multicast or unicast, responses may be synchronous or asynchronous, and network communication may be managed using a range of transport protocols, most commonly TCP or UDP.

The gateway between SLP and UPnP shown in Figure 4.3 involves a variety of these communication methods. Such a gateway may be used in a service discovery environment that provides mechanisms for dynamically discovering available services in a network. For example, a washing machine may search for a loudspeaker service and use it to play a sound once the washing is complete. In this scenario, the washing machine includes a SLP (Service Location Protocol) user agent and the speaker uses a UPnP (Universal Plug and Play) service agent to advertise its location and audio characteristics. UPnP is a wrapper for SSDP [46] and HTTP [42], which are used at different stages of the service discovery process.

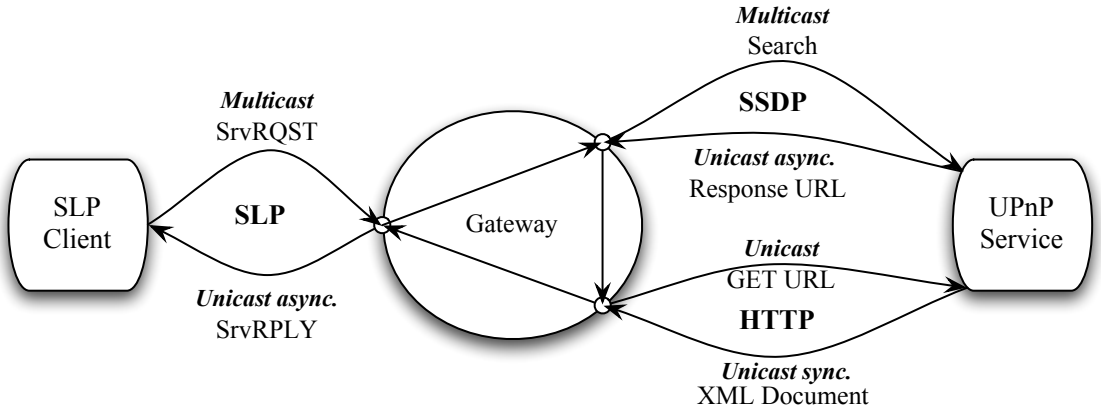


Figure 4.3: SLP to UPnP gateway

From a multicast SLP `SrvRQST` service discovery request, the SLP/UPnP gateway extracts appropriate information, such as the service type, and sends a multicast SSDP `SEARCH` request. If a service is found, the UPnP service agent asynchronously returns a unicast SSDP response containing the URL of the service description to the gateway. Then, the gateway extracts the URL and sends a unicast HTTP `GET` request to it to retrieve the service description as an XML document. Finally, the gateway extracts information from the XML document and creates an SLP `SrvRPLY` response, which it returns to the SLP client. This gateway must thus manage both multicast and unicast requests, and synchronous and asynchronous responses. Mixing the translation logic with these underlying protocol details complicates the development of the gateway code.

4.3 Specifying a Gateway Using Z2z

As illustrated in Section 4.2, a gateway receives a single request from the source device, translates it into one or a series of requests for one or more target devices, and then returns a response to the source device. It may additionally need to save some state when the source protocol has a notion of session that is different from that used by the target protocol, or when the interaction with the target device(s) produces some information that is needed by subsequent source requests.

Our case studies show that there are two main challenges to developing such a gateway: (1) manipulating messages and maintaining session state, to deal with the problem of mismatched protocol expressiveness, and (2) managing the interaction with the network, to deal with the problem of heterogeneous communication methods. In z2z, these are addressed through the combination of a DSL that allows the gateway developer to describe the translation between two or more protocols in a high-level way, and a runtime system that provides network interaction and data management facilities specific to the domain of

gateway development. We describe the DSL in this section, and present the architecture of the runtime system in the next section. The complete grammar of z2z is provided in Appendix C.

4.3.1 Overview of the z2z language

To create a gateway, the developer must provide three kinds of information: 1) how each protocol interacts with the network, 2) how messages are structured, and 3) the translation logic. To allow each kind of information to be expressed in a simple way, z2z provides a specific kind of module for each of them: protocol specification (PS) modules for defining the characteristics of protocols, message specification (MS) modules for describing the structure of protocol messages, and a message translation (MT) module for defining how to translate messages between protocols. These modules are implemented using the z2z DSL, which hides the complexities of network programming and allows specifying the relevant operations in a clear and easily verifiable way.

As illustrated in Section 4.2, a gateway may involve any number of protocols. Thus, a gateway specification may contain multiple protocol specification modules and message specification modules, according to the number of protocols and message types involved. A gateway specification always contains a single message translation module. Figure 4.4 shows the architecture of the SIP/RTSP gateway in terms of its use of these modules. We now present these modules in more detail, using this gateway as an example.

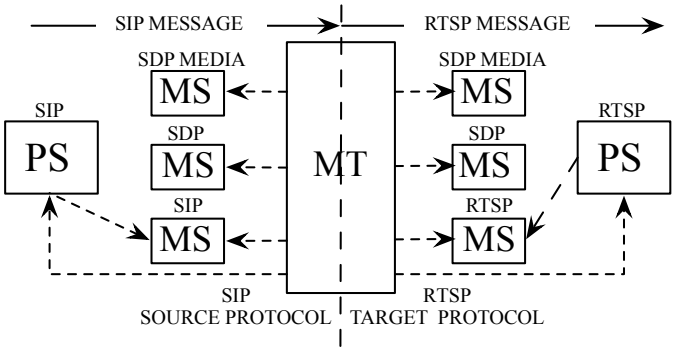


Figure 4.4: The structure of a z2z gateway specification (arrows represent dependencies)

4.3.2 Protocol specification module

The protocol specification module defines the properties of a protocol that a gateway should use when sending or receiving requests or responses. As illustrated in Figure 4.5 for the SIP protocol, this module declares the following information.

Attributes. Protocols vary in their interaction with the network, in terms of the transport protocol used, whether requests are sent by unicast or by multicast, and whether responses are received synchronously or asynchronously. The `attributes` block of the protocol specification module indicates which combination is desired. Based on this information, the runtime system provides appropriate services. For example, SIP relies on UDP (`transport` attribute, line 4), sends requests in unicast (`mode` attribute, line 5), and receives responses asynchronously. If the transmission mode is asynchronous, the protocol specification must also include a `flow` block (line 26) describing how to match requests to responses. The `transport` attribute can also be equal to `TCP` to indicate that a new TCP connection needs to be created for each request and to `TCP alive` to specify that multiple requests must be sent inside the same TCP connection.

```
1  protocol sip {
2
3    attributes {
4      transport = udp/5060;
5      mode = async/unicast;
6    }
7
8    request req {
9      response invite    when req.method == "INVITE";
10     response bye       when req.method == "BYE";
11     void ack           when req.method == "ACK";
12   }
13
14   response {
15     C1xx; C2xx; C3xx; C4xx; C5xx; C6xx;
16     CFailure = C3xx | C4xx | C5xx | C6xx;
17   }
18
19   sending request req { ... }
20
21   sending response resp (request req) {
22     resp.cseq = req.cseq;
23     resp.callid = req.callid;
24     ...
25   }
26   flow =    { callid, cseq }
27   session_flow = { callid }
28
29   automata server {
30     // INVITE server transaction      RFC3261, p125
31     automaton a1 {
32       start init : ?invite -> proceeding;
33       proceeding : !C1xx -> proceeding;
34       proceeding : !C2xx -> confirmed;
35       proceeding : !CFailure -> completed;
36       completed  : ?ack -> terminated;
37       confirmed  : ?ack -> a2.init;
38       final terminated;
39     }
40     automaton a2 {
41       init : ?bye -> trying;
42       trying : !C2xx | !CFailure -> completed;
43       trying : !C1xx -> proceeding;
44       proceeding : !C1xx -> proceeding;
45       proceeding : !C2xx | !CFailure -> completed;
46       final completed;
47     }
48     ...
49   }
50 }
```

Figure 4.5: The PS module for the SIP protocol

Requests and responses. The entry point of a gateway is the reception of a request. On receiving a request, the gateway dispatches it to the appropriate handler in the message translation module. The `request` block (lines 8-12) of the protocol specification module declares how to map messages to handlers. For each kind of request that should be handled by the gateway, the `request` block indicates the name of the handler (`invite`, `bye`, and `ack`, for SIP), whether the request should be acknowledged by a response (`response` if a response is allowed and `void` if no response is needed), and a predicate, typically defined in terms of the fields of the request, indicating whether a request should be sent to the given handler. Similarly, the `response` block (lines 14-17) defines classes of responses.

Sending. A protocol typically defines certain basic information that all messages must contain. Rather than requiring the developer to specify this information in each handler, this information is specified in a `sending` block for requests when the protocol is used as the target protocol and for responses when it is used as the source protocol of the gateway. The `sending` block for requests is parameterized by only the request, while the `sending` block for responses is parameterized by both the previous incoming request and the response, allowing elements of the response to be initialized according to information stored in the request. For example, the SIP sending block for responses copies the `cseq` and `callid` fields from the request to the response (lines 22-23). When information need to be accumulated over the treatment of several messages, the protocol specification module may declare global variables.

Flow and session_flow. When a target protocol sends responses asynchronously, an incoming response must be associated with a previous request, to restart the associated handler. The `flow` block (line 26), specifies the message elements that determine this association. In SIP, a request and its matching response have the same sequence number (`cseq`) and call id (`callid`). A `session_flow` block (line 27) similarly specifies how to recognize messages associated with a session.

The information in the protocol specification module impacts operations that are typically scattered throughout the gateway. In providing the protocol specification module as a language abstraction, we have identified the elements of the protocol definition that are relevant to gateway construction and collected them into one easily understandable unit. Furthermore, creating a protocol specification module is lightweight, involving primarily selecting properties rather than implementing their support, making it easy to incorporate many different kinds of protocols into a single gateway, as illustrated by the SLP/UPnP gateway described in Section 4.2.

Automata. Finite state machine models are typically used to describe the expected interactions (possible sequences of requests and responses) of the protocol. The `automata` block (lines 29-49) of the protocol specification module defines the sate machines of a client (sending request) and a server (receiving requests) according to the protocol. For example, the `automata` block shown from lines 29 to 49 describes the state machine of the `INVITE` server transaction, i.e., the sequence of possibles messages upon the reception of an `INVITE` request (line 32).

4.3.3 Message specification module

A network message is organized as a sequence of text lines, or of bits, for a binary protocol, containing both fixed elements and elements specific to a given message. A gateway must extract relevant elements from the received request and use them to create one or more requests according to the target protocol(s). Similarly, it must extract relevant elements from the received responses and ultimately create a response according to the source protocol. Extracting values from a message represented as a sequence of text or binary characters is unwieldy, and creating messages is even more complex, because the element values may become available at different times, making it difficult to predict the message size and layout.

In z2z, the message specification module contains a description of the messages that can be received and created by a gateway. Based on this description, the z2z compiler generates code for accessing message elements made available by an externally developed parser and inserting message elements into a created message. There is one message specification module per protocol relevant to the gateway, including both the source and target protocols, as represented by the protocol specification modules, and one per any higher level message type that can be embedded in the requests and responses. For example, the SIP/RTSP camera gateway uses not only SIP and RTSP message specification modules but also message specification modules for SDP Media and SDP, which are not associated with protocol specification modules, as SDP Media and SDP messages are embedded into SIP requests and responses.

A message specification module provides a *message view* describing the relevant elements of incoming messages and *templates* for creating new messages. The set of elements is typically specific to the purpose of the gateway, not generic to the protocol, and thus the message specification module is separate from the protocol specification module. We illustrate the declarations of the message view and the templates in the SIP message specification module used in our camera gateway.

Message view A message view describes the information derived from received messages that is useful to the gateway. It thus represents the interface between the gateway and the message parser. Z2z does not itself provide facilities for creating message parsers, but instead makes it possible to plug in one of the many existing network message parsers[2] or to construct one by hand or using a parser generator targeting network protocols, such as Zebu [21].

Because SIP is the source protocol of the camera gateway, its message view describes the information contained in a SIP request. An excerpt of the declaration of this view is shown in Figure 4.6a. It consists of a sequence of field declarations, analogous to the declaration of a C-language structure. A field declaration indicates whether the field is mandatory or optional, whether it is public or private, its type, and its name. A field is mandatory if the protocol RFC specifies that it is always present, and optional otherwise. A field is public if it can be read by the gateway logic, and private if it can only be read by the protocol specification. The type of a field is either integer, fragment, or a list of one of these types. A field of type fragment is represented as a string, but the gateway logic can cause it to be parsed as a message of another protocol, such as SDP or SDP Media, in our example.

[2] oSIP: http://www.gnu.org/software/osip
Sofia-SIP: http://opensource.nokia.com/projects/sofia-sip/
Livemedia: http://www.livemediacast.net/about/library.cfm

```
1  read {
2    mandatory private int cseq;
3    mandatory private fragment callid;
4    mandatory private fragment via;
5    mandatory private fragment to;
6    mandatory private fragment from;
7    mandatory private fragment method;
8
9    optional   private fragment to_tag;
10   optional   private int cseqsss;
11
12   mandatory public fragment uri, body;
13   mandatory public fragment from_host;
14 }
```

a) View of SIP requests

```
15 response template Invite_ok implements C2xx; {
16   magic   = "foo";
17   newline = "\r\n";
18   private fragment from, to, callid, via, contact;
19   private int cseq, content_length;
20   public fragment body, to_tag;
21 −−foo
22 SIP/2.0 200 OK
23 Via: <%via%>
24 [...]
25 Content−Length: <%content_length%>
26
27 <%body%>
28 −−foo }
```

b) Template for an INVITE method success response

Figure 4.6: SIP message specification for the camera gateway

Templates Z2z maintains messages to be created as a pair of a template view and a template. The template language is adapted from that of Repleo [5]. A message is created in the message translation module by making a new copy of the template view, and initializing its fields, in any order. At a send or return operation in the message translation module, the template representing the message is flushed, filling its holes with the corresponding values from the view.

Because SIP is the source protocol of our camera gateway, its templates describe the information needed to create SIP responses. Typically, there are multiple response templates for each method, with one template for each relevant success and failure condition. Figure 4.6b shows the template for a response indicating the success of an `INVITE` request. The `implements` keyword (line 15) is used to describe the class of responses the message created by the template belongs to. The possible classes of requests and responses are defined in the protocol specification module, see Section 4.3.2.

A template declaration has three parts: the structural declarations (lines 16-17), the template view (lines 18-20), and the template text (lines 21-28). The structural declarations indicate a string, `magic`, marking the start and end of the template text, and the line separator, `newline`, specified by the protocol RFC. The template view is analogous to the message view, except that the keywords `mandatory` and `optional` are omitted, as all fields are mandatory to create a message. The private fields are filled in by the `sending` block of the protocol specification. The public fields are filled in by the message translation module. Finally, the template text has the form of a message as specified by the protocol RFC, with holes delimited by `<%` and `%>`. These holes refer to the fields of the template view, and are instantiated with the values of these fields when the template is flushed. Binary templates, as needed for SLP messages in our service discovery gateway (Section 4.2), can be defined, using the keyword `binary`.

4.3.4 Message translation module

The message translation module expresses the message translation logic, which is the heart of the gateway. This module consists of a set of handlers, one for each kind of relevant incoming request, as indicated by the protocol specification module. Handlers are written using a C-like notation augmented with domain-specific operators for manipulating and constructing messages, for sending requests and returning responses, and for session management. Figure 4.7 shows the `invite` handler for the camera gateway.

Manipulating message data A handler is parameterized by a view of the corresponding request. The information in the view can be extracted using the standard structure field access notation (line 13). If a view element is designated as being optional in the message specification module, it must be tested using `empty` to determine whether its value is available before it is used (line 22). A view element of type `fragment` can be cast to a message type, using the usual type cast notation. In line 23, for example, the body of the request is cast to an SDP message, which is then manipulated according to its view (line 24).

```
1  fragment session_id = "";
2
3  sip response invite (sip request s) {
4    rtsp response rr;
5    sip  response sr, failed;
6    sdp_media message rtsp_m, sip_m, media_resp;
7    sdp message sdp_rtsp, sdp_sip, sdp_resp;
8    fragment list inv_medias, rtsp_medias;
9
10   // Create error response
11   failed=Invite_failure(code=400,to_tag=random());
12
13   sdp_rtsp = (sdp message)(s.body);
14   inv_medias = (fragment list)(sdp_rtsp.medias);
15
16   // Notify that something is happening
17   preturn Invite_provisional(body = "",
18                    to_tag = random());
19
20   // Retrieve the description of a media object
21   rr = send(Describe(resource = s.uri_uname));
22   if (empty(rr.body)) return failed;
23   sdp_sip = (sdp message)(rr.body);
24   rtsp_medias = (fragment list)(sdp_sip.medias);
25
26   // See whether a compatible video format exists
27   foreach (fragment rtsp_m_ = rtsp_medias) {
28     rtsp_m = (sdp_media message)rtsp_m_;
29     if (rtsp_m.type == "video") {
30       foreach (fragment sip_m_ = inv_medias) {
31         sip_m = (sdp_media message)sip_m_;
32 if ((rtsp_m.type == sip_m.type) &&
33     (rtsp_m.profile == sip_m.profile)) {
34   // Found something compatible
35   if (empty(rtsp_m.control))
36     return failed;
37   // Specify the transport mechanism
38   rr = send(Setup(uri=rtsp_m.control,
39                   destination=s.from_host,
40                   port1=sip_m.port,
41                   port2=sip_m.port+1));
42   if (empty(rr.sessionId) ||
43       empty(rr.code) || rr.code > 299)
44     return failed;
45   session_start();
46   session_id = rr.sessionId;
47   // Tell the server to start sending data
48   rr = send(Play(resource = s.uri_uname,
49                  sessionId = session_id));
50   if (empty(rr.code) || rr.code > 299) {
51     session_end(); return failed; }
52   media_resp = Media(type = sip_m.type,
53                      profile = sip_m.profile);
54   if (empty(rr.server_port))
55     media_resp.port = 0;
56   else media_resp.port = rr.server_port;
57   sdp_resp = Sdp_media(header=
58         sdp_rtsp.header,media=media_resp);
59   return Invite_ok(body = sdp_resp,
60                    to_tag = random());
61 }}}}
62 return failed; }
```

Figure 4.7: The `INVITE` handler of the message translation module for the camera gateway

A handler creates a message by invoking the name of the corresponding template (line 17). Keyword arguments can be used to initialize the various fields (lines 17-18) or the fields can be filled in incrementally (lines 54-56). A created message is maintained as a view during the execution of the handler and flushed to a network message at the point of a `send` or `return` operation.

Sending requests and returning responses A request is sent using the operator `send`, as illustrated in line 38. If the protocol specification module for the corresponding target protocol indicates that a response is expected, then execution pauses until a response is received, and the response is the result of the `send` operation. If the protocol specification indicates that no response is expected, `send` returns immediately. There is no need for the developer to break the handler up into a collection of callback functions to receive asynchronous responses, as is required in most other languages used for gateway programming. Instead, the difference between synchronous and asynchronous responses is handled by the z2z compiler, as described in Section 4.5. This strategy makes it easy to handle the case where the gateway must translate a single request from the source device into multiple requests for the target device, requiring multiple `send` operations.

If the protocol specification module indicates a return type for a handler, then the handler must return a response. This is done using `return` (line 59), which takes as argument a message and terminates execution of the handler. A provisional response, as is needed in SIP to notify the source device that a message is being treated, can be returned using `preturn` (line 17). This operator asynchronously returns the specified message, and handler execution continues.

Session management A session is a state that is maintained over a series of messages. If the protocol specification module for the source protocol declares how messages should be mapped to sessions (`session_flow`), then the message translation module may declare variables associated with a session outside of any handler. The camera gateway, for example, declares the session variable `session_id` in line 1. The message translation module initiates a session using `session_start()` (line 45). Once the session has started any modification made to these variables persists across requests within the ses-

sion, until the session is ended using `session_end()`. At this point, all session memory is freed. The SMTP/HTTP/SMTP gateways described in Section 4.2 similarly use sessions to maintain the TCP connection identifier across multiple requests.

4.4 Formal Semantics

We have formally specified the semantics of the message translation module of z2z, enabling a precise definition of the message translation logic and its interaction with the runtime system. In this section, we describe the dynamic semantics of a subset of z2z.

4.4.1 Abstract Syntax

For conciseness, we focus on a subset of the message translation module of z2z related to operations for sending requests and returning responses, noted MTL_{send} . The abstract syntax of MTL_{send} is presented in Figure 4.8. We consider a handler that describes the translation of a message specified by a host a using some protocol A to a host b using some protocol B. We assume that the handler returns a response to host a. We now describe the relationship between a z2z handler and its MTL_{send} counterpart.

handler ::= `method(A request` *id*`)` `{` *decl** *stmt** `}`
decl ::= *type* *id*`;`
type ::= *ftype* | `A response` | `B request` | `B response`
ftype ::= `int` | `fragment` | `P message` | *ftype* `list`
stmt ::= id_1 `=` `send(`id_2`,` *sync*`);`
| `return` *id*`;` | `preturn` *id*`;`
| `rearm;`
| id_1 `=` `new(`id_2`);`
| *id* `=` *expr*`;`
| `fill(`id_1`,`id_2`,` *expr*`);`
| `if (`*expr*`)` $stmt_1$ `else` $stmt_2$
| `while (`*expr*`)` *stmt*
| `{` *stmt** `}`
sync ::= `sync` | `async`
expr ::= *"..."* | *integer* | *id*
| `access(`id_1`,` id_2`)`
| `empty(`id_1`,` id_2`)`
| $expr_1$ `==` $expr_2$ | $expr_1$ `+` $expr_2$ | $expr_1$ `-` $expr_2$

Figure 4.8: Abstract syntax of the MTL_{send} language

Sending a request. The send operation of z2z takes as parameter a request to be sent to host b and may receive a response from b as a return value. We assume in the following that `send` always returns a value. The first parameter of `send` is an identifier of type *`B request`* and refers to the message to be sent to host b. In addition, we introduce a second parameter to `send` to indicate whether the message send is synchronous (`sync`) or asynchronous (`async`). The choice of a synchronous or asynchronous send is in practice determined by the protocol (see Section 4.3.2). If the second argument to `send` is `sync`, then the gateway blocks until the response arrives. Otherwise, the thread treating the current message pauses until the response is received.

Returning a response. The `return` operator takes as its only argument an identifier *id* of type `A` *response*. The thread treating the current message returns *id* to host *a* and aborts. The `preturn` operator behaves like `return` except that the thread treating the current message returns *id* to host *a* but does not pause, block or abort. In addition to these operators, the `rearm` operator restarts the most recently started handler. This is used on receiving intermediate responses in an asynchronous protocol, i.e. the same kinds of responses that are generated using `preturn`.

Manipulating message data. In z2z, a message is created by invoking the name of the corresponding template. Keyword arguments can be used to initialize the various fields, or the fields can be filled in incrementally. In MTL_{send} , a message is created by the `new` operator that takes as argument an identifier id_2 of type *B request*, *A response* or *P message*. The statement creates a copy of the request or response template that is ready to be instantiated using the operator `fill`. The first argument of `fill` is an identifier of type *B request*, *A response*, or *P message*, previously initialized to the result of calling `new`. The second argument indicates a field name within this request, response, or message, and the last argument must be a value that is appropriate to be placed in this field.

As described in Section 4.3.4, an optional view element in z2z must be tested using `empty` to determine whether its value is available before it is used. The `empty` operator in MTL_{send} takes two arguments: an identifier of type *A request*, *B response*, or *P message* and an identifier indicating a field name within this request, response, or message.

Control flow constructions and expressions. MTL_{send} reuses z2z constructions for iterations, conditionals and sequences. In z2z, however, iterations are restricted to the use of the `foreach` statement in order to prevent infinite loops. In MTL_{send} , `foreach` is translated into `while`. MTL_{send} expressions are similar to z2z expressions. For conciseness, we focus only on three binary operators: ==, +, and −. The z2z language also supports the unary and binary operators `!`, `!=`, `&&`, `||`, `>=`, `>`, `<`, and `<=`. Because `send` does not occur in expressions but only occurs at the statement level, the issue of synchronicity only affects the semantics of statements and of terms that contain statements as subterms, *i.e.* handlers.

Abstracting the message translation module of z2z. Transforming a z2z handler into its MTL_{send} counterpart is straightforward. This is illustrated by the `BYE` handler of the SIP/RTSP camera gateway shown in Figure 4.9-a and its MTL_{send} counterpart is displayed in 4.9-b.

```
sip response bye (sip request s) {
  rtsp response rr;
  sip  response sr;

  rr = send(Teardown(resource = s.uri_username,
                     sessionId = session_id));

  sr = Bye_resp(to_tag = "");
  if (empty(rr.code))
     sr.code = 400;
  else
     sr.code = rr.code;
  session_end();
  return sr;
}
```

```
method (sip request s) {
  rtsp response rr;
  rtsp request tmp;
  sip  response sr;

  tmp = new(Teardown);
  fill(tmp, ressource,
               access(s,uri_username));
  fill(tmp, sessionId, session_id);

  rr = send(tmp, sync);

  sr = new(Bye_resp);
  fill (sr, to_tag, "");
  if (empty(access(rr, code)))
     fill(sr, code, 400);
  else
     fill(sr, code, access(rr, code));
  return sr;
}
```

Figure 4.9: BYE handler of the camera gateway service in z2z and its MTL_{send} counterpart

4.4.2 Dynamic Semantics

The small step semantics of MTL_{send} terms is described using a continuation-based abstract machine [1], similar to that used by the semantics of SPL presented in Chapter 2. Execution in this machine is specified as a sequence of configurations, starting with a configuration representing the receipt of a message and ending with a configuration representing the returning of some information to the runtime system. Intermediate configurations represent the execution of a term or the invocation of a continuation. Whenever the semantics begins the execution of a term, it adds a frame to the continuation storing all of the information required to continue execution from the point of that term. This approach makes each configuration self-contained, and is used in the semantics of asynchronous send.

The values are booleans, fragments, messages, requests and responses. `null` is both a request, a response, and a message. We assume that the rutime system provides the following functions:

- send_sync : B request $\rightarrow$ B response
- send_async : B request $\rightarrow$ cont$_{in}$ $\rightarrow$ cont $\rightarrow$ env $\rightarrow$ id $\rightarrow$ unit
- return : A response $\rightarrow$ unit
- preturn : A response $\rightarrow$ unit
- new : (A response + B request + P message) $\rightarrow$ (A response + B request + P message)
- fill : (A response + B request + P message) $\rightarrow$ id $\rightarrow$ fragment $\rightarrow$ unit
- access : (A response + B request + P message) $\rightarrow$ id $\rightarrow$ fragment
- empty : (A response + B request + P message) $\rightarrow$ id $\rightarrow$ bool

The configurations used in the semantics are as follows, where ρ is an environment mapping variables to values, v is a value other than `null`, and κ_{in} and κ are continuations, which are a list of statements. The continuation κ_{in} refers to the complete list of statements defining a handler and is used to restart the semantics while processing a `rearm` statement.

- handlers: $v \models_{\text{P}} handler$
- Declarations: $\rho \models_{\text{D}} declaration \Rightarrow \rho'$
- Expression: $\rho \models_{\text{E}} expression \Rightarrow v$
- Statements: $\rho, \kappa_{in}, \kappa \models_{\text{S}} stmt$
- Continuations: $\rho, \kappa_{in} \models_{\text{C}} \kappa$

The semantic rules are described as inference rules, with a sequence of premises above a horizontal bar, and the current configuration and the next configuration, if any, in the execution sequence below the bar, separated by an arrow ($\Rightarrow$).

Handlers:

$$\frac{\rho_0 = [id \mapsto v] \qquad \rho_0 \models_{\text{E}} decl_1 \Rightarrow \rho_1 \qquad \ldots \qquad \rho_{n-1} \models_{\text{E}} decl_n \Rightarrow \rho_n}{\begin{array}{c} v \models_{\text{P}} \textsf{method}\,(\texttt{A request } id)\ \{\ decl_1 \ldots decl_n\ \ stmt_1 \ldots stmt_m\ \} \\ \Rightarrow \rho_{n-1}, \langle stmt_1, \ldots, stmt_m \rangle, \langle\,\rangle \models_{\text{S}} \langle stmt_1, \ldots, stmt_m \rangle \end{array}}$$

Declarations
The semantics of declarations is defined as below. Each rule $\rho[id \mapsto \texttt{null}]$ adds to the environment ρ an entry for the identifier *id* with an undefined value.

$$\rho \models_{\text{D}} \texttt{fragment}\ \ id\texttt{;} \Rightarrow \rho[id \mapsto \texttt{null}]$$

$$\rho \models_{\text{D}} \texttt{A response}\ \ id\texttt{;} \Rightarrow \rho[id \mapsto \texttt{null}] \qquad \rho \models_{\text{D}} \texttt{P message}\ \ id\texttt{;} \Rightarrow \rho[id \mapsto \texttt{null}]$$

$$\rho \models_{\text{D}} \texttt{B request}\ \ id\texttt{;} \Rightarrow \rho[id \mapsto \texttt{null}] \qquad \rho \models_{\text{D}} \texttt{B response}\ \ id\texttt{;} \Rightarrow \rho[id \mapsto \texttt{null}]$$

Expressions
In the following semantic rules for expressions, we consider only the equality binary operator. Rules for other binary operators are similar and are standard since the environment ρ cannot be modified in an expression.

$$\rho \models_{\text{E}} \text{"..."} \Rightarrow \text{"..."} \qquad \rho \models_{\text{E}} id \Rightarrow \rho(id) \qquad \rho \models_{\text{E}} \texttt{empty(}id_1\texttt{,}\ id_2\texttt{)} \Rightarrow \mathsf{empty}(\rho(id_1), \rho(id_2))$$

$$\frac{v = \rho(id_1) \qquad v' = \rho(id_2) \qquad \neg\, \mathsf{empty}(v_1, v_2)}{\rho \models_{\text{E}} \texttt{access(}id_1\texttt{,}\ \ id_2\texttt{)} \Rightarrow \mathsf{access}(v_1, v_2)} \qquad \frac{\rho \models_{\text{E}} expr_1 \Rightarrow v_1 \qquad \rho \models_{\text{E}} expr_2 \Rightarrow v_2}{\rho \models_{\text{E}} expr_1\ \texttt{==}\ expr_2 \Rightarrow v_1 = v_2}$$

Statements The semantics of statements is defined below. Following a call to $\mathsf{send\_async}(r, \kappa_{in}, \kappa, \rho, id)$ a response v should ultimately be received. At this point, the runtime system creates the configuration $\rho[id \mapsto v], \kappa_{in} \models \kappa$ and restarts the semantics. The rule for `rearm` restarts the semantics at the entry point of the handler using the current environment ρ. This is done using the continuation κ_{in} that represents the body of the handler, and which is passed through each statement rule.

$$\frac{v = \rho(id_2) \qquad \mathsf{send\_sync}(v) = v'}{\rho, \kappa_{in}, \kappa \models_{\text{S}} id_1\ \texttt{=}\ \texttt{send(}id_2\texttt{,}\ \mathsf{sync}\texttt{);} \Rightarrow \rho[id_1 \mapsto v'], \kappa_{in} \models_{\text{C}} \kappa}$$

$$\frac{v = \rho(id_2) \qquad \mathsf{send\_async}(v, \kappa_{in}, \kappa, \rho, id_1)}{\rho, \kappa_{in}, \kappa \models_{\text{S}} id_1\ \texttt{=}\ \texttt{send(}id_2\texttt{,}\ \mathsf{async}\texttt{);}}$$

$$\frac{v = \rho(id) \qquad \mathsf{return}(v)}{\rho, \kappa_{in}, \kappa \models_{\text{S}} \texttt{return}\ \ id\texttt{;}} \qquad \frac{v = \rho(id) \qquad \mathsf{return}(v)}{\rho, \kappa_{in}, \kappa \models_{\text{S}} \texttt{preturn}\ \ id\texttt{;} \Rightarrow \rho, \kappa_{in} \models_{\text{C}} \kappa}$$

$$\frac{}{\rho, \kappa_{in}, \kappa \models_{\text{S}} \texttt{rearm;} \Rightarrow \rho, \kappa_{in}, \langle\rangle \models_{\text{S}} \kappa_{in}}$$

$$\rho, \kappa_{in}, \kappa \models_{\text{S}} id_1\ \texttt{=}\ \texttt{new(}id_2\texttt{);} \Rightarrow \rho[id_1 \mapsto \mathsf{new}(id_2)], \kappa_{in} \models_{\text{C}} \kappa$$

$$\frac{\rho \models_{\text{S}} expr \Rightarrow v' \qquad \mathsf{fill}(\rho(id), \rho(fld), v')}{\rho, \kappa_{in}, \kappa \models_{\text{S}} \texttt{fill(}id\texttt{,}\ \ fld\texttt{,}\ \ expr\texttt{);} \Rightarrow \rho, \kappa_{in} \models_{\text{C}} \kappa} \qquad \frac{\rho \models_{\text{S}} expr \Rightarrow v}{\rho, \kappa_{in}, \kappa \models_{\text{S}} id\ \texttt{=}\ expr\texttt{;} \Rightarrow \rho[id \mapsto v], \kappa_{in} \models_{\text{C}} \kappa}$$

$$\frac{\rho \models_{\mathrm{s}} \mathit{expr} \Rightarrow \mathsf{true}}{\rho, \kappa_{in}, \kappa \models_{\mathrm{s}} \texttt{if}\ \ (\mathit{expr})\ \ \mathit{stmt}_1\ \ \texttt{else}\ \ \mathit{stmt}_2 \Rightarrow \rho, \kappa_{in}, \kappa \models_{\mathrm{s}} \mathit{stmt}_1}$$

$$\frac{\rho \models_{\mathrm{s}} \mathit{expr} \Rightarrow \mathsf{false}}{\rho, \kappa_{in}, \kappa \models_{\mathrm{s}} \texttt{if}\ \ (\mathit{expr})\ \ \mathit{stmt}_1\ \ \texttt{else}\ \ \mathit{stmt}_2 \Rightarrow \rho, \kappa_{in}, \kappa \models_{\mathrm{s}} \mathit{stmt}_2}$$

$$\frac{\rho \models_{\mathrm{s}} \mathit{expr} \Rightarrow \mathsf{true}}{\rho, \kappa_{in}, \kappa \models_{\mathrm{s}} \texttt{while}\ \ (\mathit{expr})\ \ \mathit{stmt} \Rightarrow \rho, \texttt{while}\ \ (\mathit{expr})\ \ \mathit{stmt} :: \kappa \models_{\mathrm{s}} \mathit{stmt}}$$

$$\frac{\rho \models_{\mathrm{s}} \mathit{expr} \Rightarrow \mathsf{false}}{\rho, \kappa_{in}, \kappa \models_{\mathrm{s}} \texttt{while}\ \ (\mathit{expr})\ \ \mathit{stmt} \Rightarrow \rho, \kappa_{in} \models_{\mathrm{c}} \kappa}$$

$$\rho, \kappa_{in}, \kappa \models_{\mathrm{s}} \{\ \ \mathit{stmt}_1\ \ \ldots\ \ \mathit{stmt}_n\ \ \} \Rightarrow \rho, \kappa_{in} \models_{\mathrm{c}} \langle \mathit{stmt}_1, \ldots, \mathit{stmt}_n \rangle @ \kappa$$

$$\rho, \kappa_{in} \models_{\mathrm{c}} \mathit{stmt} :: \kappa \Rightarrow \rho, \kappa_{in}, \kappa \models_{\mathrm{s}} \mathit{stmt}$$

4.5 Implementation

Our implementation of the z2z gateway generator comprises a compiler for the z2z language and a runtime system. From the z2z specification of a gateway, the z2z compiler generates C code that can then be compiled using a standard C compiler and linked with the runtime system. The generated code is portable enough to run on devices ranging from desktop computers to constrained devices such as PDAs or home appliances. The runtime system defines various utility functions and amounts to about 7500 lines of C code. The z2z compiler is around 10500 of OCaml code. The compiler can be used offline to produce the gateway code and therefore is not required to be present on the gateway device. We first describe the verifications performed by the compiler, then present the main challenges in code generation, and finally present the runtime system.

4.5.1 Verifications

The z2z compiler performs consistency checks, dataflow analyses and model checking to detect erroneous specifications and to ensure the generation of safe gateway code. These analyses are based on the formal semantics of z2z presented in Section 4.4.

Consistency checks. As was shown in Figure 4.4, there are various dependencies between the modules making up a z2z gateway. The z2z compiler performs a number of consistency checks to ensure that the information declared in one module is used elsewhere according to its declaration. The main inter-module dependencies are derived from the `request` and `sending` blocks of the protocol specification and the types and visibilities of the elements of the message views. The `request` block associated with the source protocol declares how to dispatch incoming requests to the appropriate handlers and whether a response is expected from these handlers. The z2z compiler checks that the message translation module defines a handler for each kind of message that should be handled by the gateway and that each handler has an appropriate return type. The `sending` block of a protocol specification module initializes some fields for all requests or responses sent using that protocol. The compiler checks that every template view defined in the corresponding message specification module includes all of these fields. Finally, the message specification module indicates for each field of a view the type of value that the field can contain and whether the field can be accessed by the message translation module (`public`) or only by the protocol

specification module (`private`). The z2z compiler checks that the fields are only used in the allowed module and that every access or update has the declared type.

Dataflow analyses. The z2z compiler performs a dataflow analysis within the message translation module to ensure that values are well-defined when they are used. The principal issues are in the use of optional message fields, session variables, and created messages. A message specification module may declare some message fields as `optional`, indicating that they may be uninitialized. The z2z compiler enforces that any reference to such a field is preceded by an `empty` check. Session variables cannot be used before a `session_start` operation or after a `session_end` operation. The z2z compiler checks that references to these variables do not occur outside these boundaries. Finally, the z2z compiler checks that all public fields of a template are initialized before the template is passed to `send` and that all execution paths through the `sending` block of the corresponding protocol specification module initialize all `private` fields.

Model checking. The z2z compiler checks that the gateway returns responses and send requests according to the state machines of the source and target protocols that describe the sequences of messages that can be received and send. In addition, the z2z compiler enforces the valid usage of sessions and of the TCP protocol. Section 4.6 describes in details the model checker provided by z2z for verifying protocol correctness.

4.5.2 Code generation

The main challenges in generating code from a z2z specification are the implementation of the `send` operation, the implementation of the variables used by the message translation module, and the implementation of memory management.

The `send` operation. The handlers of a z2z message translation module are specified as sequential functions, with `send` having the syntax of a function call that may return a value. If the target protocol returns responses synchronously, the z2z compiler does indeed implement `send` as an ordinary function call. If the target protocol returns responses asynchronously, however, this treatment is not sufficient. In this case, the implementation of `send` does not return a value, but must instead receive as an argument information about the rest of the handler so that the handler can be restarted when a response becomes available. The standard solution is to decompose the code into a collection of callback functions, which are tedious, error-prone, and unintuitive to write by hand. Fortunately, it has been observed that such callback functions amount to *continuations*, which can be created systematically [55]. The z2z compiler thus splits each handler at the point of each `send` to create a collection of functions, of which the first represents the entry point of the handler and the rest represent some continuation.

Figure 4.10 illustrates the splitting of a handler performed by the z2z compiler when the target protocol of a `send` returns responses asynchronously. This code contains three `send` operations, one on line 6 and the others in each of the if branches (lines 12 and 18). The continuation function for the `send` on line s1 contains the code in the region labeled (2). The continuation function for the `send` on line s2 contains the code in region (3) and the one for the `send` on line s3 contains the code in region (4). As shown, the latter two continuation functions explicitly contain only the code within the corresponding if branch. The execution of the handler must, however, continue to the code following the if statement, which is in the continuation of both `send` operations. To reduce the code size, the z2z compiler factorizes the code after the if statement into a separate continuation function, labeled (5), which is invoked by both (3) and (4) after executing the if branch code [93].

Variables. Splitting a handler into a set of disjoint continuation functions in the asynchronous case complicates the implementation of the handler's local variables when these variables are used across `send`s

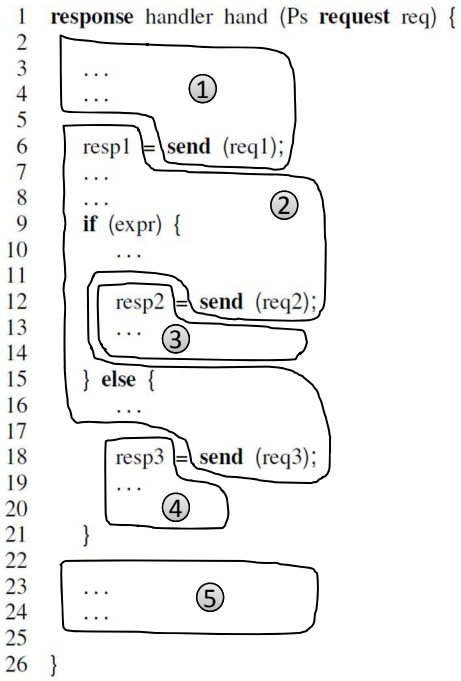


Figure 4.10: Code slicing for continuations

and thus by multiple continuations. The z2z compiler identifies handler variables whose values must be maintained across asynchronous `send`s, and implements them as elements of an environment structure that such a `send` passes to the runtime system. The runtime system stores this environment, and passes it back to the stored continuation function when the corresponding response is received.

Session variables are similarly always implemented in an environment structure, as by design they must be maintained across multiple invocations of the message translation module. Between handler invocations, the runtime system stores this environment with the other information about the session.

Dynamic memory management. Template constructors dynamically allocate memory, analogous to Java's `new` operation. This memory may be referenced from arbitrary local and session variables of the message translation module, and must be freed when no longer useful. Had we translated z2z into a garbage collected language such as Java, then we could rely on the garbage collector to free this memory. We have, however, used C, to avoid the overhead of including a fully-featured runtime system such as the JVM. To avoid the risk of dangling pointers, the z2z compiler generates code to manage reference counts [26]. An alternative, for future work, is to use a garbage collector for C code [9].

4.5.3 Runtime system

The z2z runtime system implements a network server capable of simultaneously handling many messages, that may rely on various protocols. The server is parameterized by the information specific to each protocol, as provided in the corresponding protocol specification module (Section 4.3.2). When a message is received, the runtime system invokes the corresponding parser to construct a message view as defined in the message specification module. The runtime system then executes the code generated from the message translation module for the handler corresponding to the incoming request. The runtime system also provides various utility functions to send requests synchronously or asynchronously, to save and restore environments, to manage sessions, and to perform various other operations.

Receiving network messages The z2z runtime system is designed to efficiently juggle many incoming requests simultaneously. It is multithreaded, based on the use of a single main thread and a pool of worker threads. The main thread detects an incoming connection, and then assigns the processing of this connection to an available worker thread. The pool of worker threads avoids the high overhead that would be entailed by spawning a new thread per connection, and thus contributes to the overall efficiency of the approach. It furthermore avoids the mutex contention that would be incurred by the use of global shared variables. The z2z developer does not need to be aware of these details.

Receiving messages via TCP poses further challenges. In this case, a stream of messages arrives within a single connection. Depending on the protocol, substantial computation may be required to isolate the individual messages within a stream. Z2z does not itself provide facilities for creating stream-oriented message parser but instead makes it possible to plug in externally developed code. To avoid dropping messages, the main thread assigns two worker threads to a TCP connection. One, the *producer*, receives data from the incoming TCP stream and separates it into messages, while the other, the *consumer*, applies the gateway logic. These threads communicate via shared memory. When the producer has extracted a message from the incoming stream, it sends a signal to the consumer, which then reads the message in the shared memory and processes it. On completion, it sends a signal to the producer. This approach allows the main thread to multiplex I/O on a set of server sockets to provide gateway service to multiple devices.

Network protocol gateways furthermore must manage multiple concurrent connections between many devices. This adds significantly to the complexity of the gateway implementation. For example, to communicate with a device that uses SMTP, multiple requests must be sent inside the same TCP connection, while for a RTSP device each request requires the creation of a new TCP connection (see Section 4.3.2). The z2z runtime system hides these details from the gateway developer. As illustrated by the HTTP/SMTP example, the runtime system must keep open the TCP connection used to send SMTP requests even if incoming requests are responded to asynchronously. However, in this case, subsequent incoming requests are not related to each other and the runtime system needs to know which TCP connection to use. To address this issue, the runtime system maintains a table of current active TCP connections and provides references to them, so that they can be retrieved later. The runtime system can also seamlessly switch from IPv4 to IPv6, send messages in unicast or multicast, and use UDP or one or many TCP connections, as specified by attributes in the protocol specification module, without requiring any additional programming from the gateway developer.

Processing a message When a thread is assigned the processing of a message, it executes the message parser of the corresponding protocol to construct a message view, as described in Section 4.3.3. Then, it calls the `dispatch` function, generated by the z2z compiler from the PS module, to select the handler to execute. If a handler sends requests asynchronously, the runtime system explicitly suspends the control flow and saves the current continuation, handler state, and session state in a global shared memory. The local memory allocated for the current thread is freed and the thread returns to the main pool. When a response is received by the main thread, the runtime system assigns its processing to an available worker thread, restores the corresponding states and continuation, and continues the execution of the handler.

4.6 Model Checking

As described in Section 4.5.1, the z2z compiler statically performs a number of consistency checks such that a handler accepts only requests from the source protocol. In addition, z2z includes a model checker to check that the gateway returns responses and send requests according to the state machines of the source and target protocols. The z2z model checker also enforces the valid usage of sessions and of the TCP protocol. To this end, the model checker constructs an automaton from the z2z code of the gateway that models the exchange of messages between the gateway and the source and target devices. Automata indicating the constraints on the source and target protocols are also constructed from the z2z protocol specifications. These automata are then combined, along other automata describing the correct usage of sessions and TCP,

to check correctness properties of the gateway. In the remainder of this section, we describe how these automata are constructed and the checking process.

4.6.1 Modeling MTL code

For concisness, we consider a simplified MTL language (see Figure 4.11) consisting of the `send` and `return` statements as well as `session_start` and `session_end` statements for session initiating and ending. The treatment of other kinds of statements such as operations on TCP connections is similar. In this language, $expr$ in an arbitrary expression. $\mathsf{SV}(expr)$ returns the set of session variables in $expr$. In a `send` or `return` statement, L is a list of all of the kinds of messages that may be sent or returned. This information is not explicit in the source code, but is made available by the MTL compiler at the AST level.

$$
\begin{array}{rcl}
\textit{stmt} & ::= & \texttt{send(}L\texttt{)} \\
& | & \texttt{return(}L\texttt{)} \\
& | & \texttt{session\_start()} \\
& | & \texttt{session\_end()} \\
& | & \texttt{if (}\textit{expr}\texttt{)}\ \textit{stmt}_1\ \texttt{else}\ \textit{stmt}_2 \\
& | & \texttt{\{}\ \textit{stmt}_1\ \textit{stmt}_2\ \texttt{\}}
\end{array}
$$

Figure 4.11: Simplified MTL language

A node in the automaton modeling a MTL handler is defined by a name and a list of pairs of a message type (or epsilon) and a destination node. The possible messages are shown in Figure 4.12.

Interaction between the client and the gateway	Accept(*message_name*) Return(*message_name*)
Interaction between the gateway and a server	Send(*message_name*) Receive(*message_name* + ε)
Sessions	SessionStart SessionEnd SessionAccess
TCP	TCPConnect TCPGetConnection TCPUseConnection TCPClose

Figure 4.12: Possible messages in MTL automaton

The rules for constructing an automaton from a MTL handler are described as inference rules with a sequence of premises above a horizontal bar and a judgment below the bar (see Figure 4.13). These rules use the following functions for constructing an automaton. The function *create_node()* creates a node in the automaton. The function *connect(start,end,l)* adds to the node *start* an edge labelled l that leads to the node *end*.

The construction of an automaton for a statement S is defined in terms of a judgment $out \vdash^S S \rightarrow (start, end)$, where *start* is the start node of the automaton associated with S and *end* is the unique end node. The construction of an automaton for an expression E is defined in terms of a judgment $\vdash^E E \rightarrow (start, end)$, where *start* is the start node of the automaton associated with E and *end* is the unique end

$$\frac{\textit{start} = \textit{create\_node()} \quad \textit{middle} = \textit{create\_node()} \quad \textit{end} = \textit{create\_node()} \quad \forall l \in L : \textit{connect(start,middle,}\mathsf{Send}\textit{(l))} \quad \textit{connect(middle,end,}\mathsf{Receive}(\varepsilon)\textit{)}}{\textit{out} \vdash^S \texttt{send}\,(L) \rightarrow \textit{start}, \textit{end}}$$

$$\frac{\textit{start} = \textit{create\_node()} \quad \textit{end} = \textit{create\_node()} \quad \forall l \in L : \textit{connect(start,out,}\mathsf{Return}\textit{(l))}}{\textit{out} \vdash^S \texttt{return}\,(L) \rightarrow \textit{start}, \textit{end}}$$

$$\frac{\textit{start} = \textit{create\_node()} \quad \textit{end} = \textit{create\_node()} \quad \textit{connect(start,end,}\mathsf{SessionStart}\textit{)}}{\textit{out} \vdash^S \texttt{session\_start}\,() \rightarrow \textit{start}, \textit{end}}$$

$$\frac{\textit{start} = \textit{create\_node()} \quad \textit{end} = \textit{create\_node()} \quad \textit{connect(start,end,}\mathsf{SessionEnd}\textit{)}}{\textit{out} \vdash^S \texttt{session\_end}\,() \rightarrow \textit{start}, \textit{end}}$$

$$\frac{\begin{array}{c}\textit{out} \vdash^S \textit{stmt}_1 \rightarrow \textit{start}_1, \textit{end}_1 \quad \textit{out} \vdash^S \textit{stmt}_2 \rightarrow \textit{start}_2, \textit{end}_2 \\ \textit{start} = \textit{create\_node()} \quad \textit{end} = \textit{create\_node()} \quad \textit{connect(start,start}_1\textit{,}\varepsilon\textit{)} \quad \textit{connect(start,start}_2\textit{,}\varepsilon\textit{)} \\ \textit{connect(end}_1\textit{,end,}\varepsilon\textit{)} \quad \textit{connect(end}_2\textit{,end,}\varepsilon\textit{)} \quad \vdash^E E \rightarrow \textit{start}_e, \textit{end}_e \quad \textit{connect(end}_e\textit{,start,}\varepsilon\textit{)}\end{array}}{\textit{out} \vdash^S \texttt{if}\ \ (E)\ \ \textit{stmt}_1\ \ \texttt{else}\ \ \textit{stmt}_2 \rightarrow \textit{start}_e, \textit{end}}$$

$$\frac{\textit{out}_1 \vdash^S \textit{stmt}_1 \rightarrow \textit{start}_1, \textit{end}_1 \quad \textit{out}_2 \vdash^S \textit{stmt}_2 \rightarrow \textit{start}_2, \textit{end}_2 \quad \textit{connect(end}_1\textit{,start}_2\textit{,}\varepsilon\textit{)}}{\textit{out} \vdash^S \{\ \ \textit{stmt}_1\ \ \textit{stmt}_2\ \ \} \rightarrow \textit{start}_1, \textit{end}_2}$$

$$\frac{\textit{start} = \textit{create\_node()} \quad \textit{end} = \textit{create\_node()} \quad \mathsf{SV}(\textit{expr}) = \emptyset \quad \textit{connect(start,end,}\varepsilon\textit{)}}{\vdash^E \textit{expr} \rightarrow \textit{start}, \textit{end}}$$

$$\frac{\textit{start} = \textit{create\_node()} \quad \textit{end} = \textit{create\_node()} \quad \mathsf{SV}(\textit{expr}) \neq \emptyset \quad \textit{connect(start,end,}\mathsf{SessionAccess}\textit{)}}{\vdash^E \textit{expr} \rightarrow \textit{start}, \textit{end}}$$

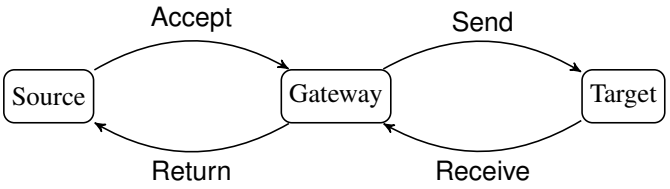
Figure 4.13: Rules for creating an automaton from MTL code

node. For a complete handler H for method h, the automaton with start node *start* is constructed according to the following rule:

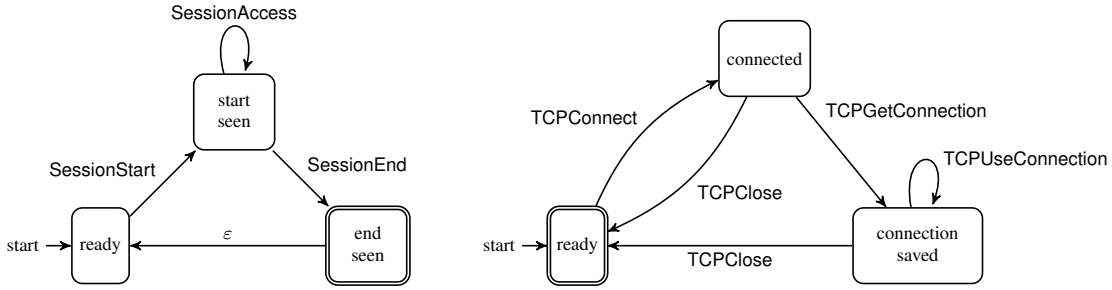
$$\frac{\textit{start} = \textit{create\_node()} \quad \textit{middle} = \textit{create\_node()} \quad \textit{out} = \textit{create\_node()} \quad \textit{connect(start,middle,}\mathsf{Accept}\textit{(h))} \quad \textit{out} \vdash^S H \rightarrow \textit{start}_H, \textit{end}_H \quad \textit{connect(middle,start}_H\textit{,}\varepsilon\textit{)}}{\vdash^H H \rightarrow \textit{start}}$$

4.6.2 Modeling protocols

We consider the case of a gateway G from a source protocol P_s to a target protocol P_t. For each of P_s and P_t we have an automaton A_{P_s} and an automaton A_{P_t}, respectively, indicating the constraints on the messages that they are able to receive and their behaviors on these messages. These automata are described in the protocol specification module, as described in Section 4.3.2. For each message m recognized by P_s, the gateway has a handler G_m. These handlers are connected within and infinite loop. That is, the automaton A_G for the gateway can be viewed as being represented by the regular expression $(\mathsf{G}_{m_1} \mid \ldots \mid \mathsf{G}_{m_n})^*$. The messages exchanged by the source and target protocols and the gateway are as follows:



In addition to the automata A_{P_s} and A_{P_t}, the z2z model checker uses an automaton $A_{sessions}$ representing the valid usage of sessions. The z2z model checker also uses an automaton A_{tcp} representing the valid usage of the TCP protocol. These automata do not depend on the specification of the gateway, and are therefore provided by the z2z model checker as builtin information.



4.6.3 Algorithm

Although the MTL automaton contains all edges, we need to specialize it for the needs of individual checks. The model checking algorithm relies on several operators for adjusting the set of edges in an automaton. We represent an automaton abstractly as a collection of nodes N and a collection of directed edges $E = N \times L \times N$, where (n_1, l, n_2) is an edge from node n_1 to node n_2 with label l. The operators are then defined on an automaton (N, E) as follows:

- augment_aut$((N, E), L) = (N, E \cup \{(n, l, n) \mid n \in N \wedge l \in L\})$. This function is used to extend an automaton such that it allows any number of trannsitions for any message l in L at any state n in N.
- determinize$((N, E), L)$. This function creates a deterministic version of $(N \cup \{\text{sink}\}, E \cup \{(n, l, \text{sink}) \mid n \in N \wedge l \in L\})$. The argument L is used here to extend the set of edge labels of the automaton so that it can be combined with another automaton with the larger set of edge labels.
- filter_aut$((N, E), \phi, L) =$ determinize$(N, \{(n_1, l, n_2) \mid (n_1, l, n_2) \in E \wedge \phi(l)\}, L)$. This function is used to limit an automaton to edges whose labels satisfy the property ϕ. The elements of L are assumed to satisfy this property. We provide four filters as described below

$$\begin{aligned}
\phi_{P_s}(l) &= l \in \{\text{Accept}, \text{Return}\} \\
\phi_{P_t}(l) &= l \in \{\text{Send}, \text{Receive}\} \\
\phi_{session}(l) &= l \in \{\text{SessionStart}, \text{SessionAccess}, \text{SessionEnd}\} \\
\phi_{tcp}(l) &= l \in \{\text{TCPConnect}, \text{TCPGetConnection}, \text{TCPUseConnection}, \text{TCPClose}\}
\end{aligned}$$

Verification setup

The verification process takes three automata as arguments: A_{P_s} for the source protocol, A_G for the gateway, and A_{P_t} for the target protocol (we assume only one target protocol for conciseness). The verification process also refers to the automata $A_{session}$ and A_{tcp} shown previsouly.

The first step is to replace all Receive messages in the A_{P_t} by Receive(ϵ) messages that do not specify the type of message received. This is due to the fact that MTL provides no support for making the type of message received apparent.

The next step is to collect the various kinds of edge labels in the various automata:

- left_edges: The edges in A_{P_s}.
  These are Accept and Return edge labels.
- mtl_edges: The edges in the A_G except the Accept edges.
  These are Return, Send and Receive(ϵ) edge labels, and the various session and TCP edge labels.
- right_edges: The edges in A_{P_t}.
  These are Send and Receive(ϵ) edge labels.
- session_edges: The edges in $A_{sessions}$.
  These are SessionStart, SessionEnd, and SessionAccess.
- TCP_edges: The edges in A_{tcp}.
  These are TCPConnect, TCPClose, TCPGetConnection, and TCPUseConnection.

Gateway automaton

The MTL code consists of a set of handlers, with no indication of the order in which they may be executed. This order is instead determined by the source protocol. To restrict the gateway automaton to the sequences of messages that can be received from the source protocol, we essentially want to intersect the source protocol automaton with the gateway automaton. A straightforward intersection, however, will fail, because the gateway automaton contains the edges for the interaction with the target protocol and possibly with sessions and TCP, that are not in the source protocol automaton. Nevertheless, we would like to preserve these edges in the result of the intersection, so that this automaton can be checked against the target protocol automaton, the session automaton, and the TCP automaton. As the source protocol has no preference about when and how many of these extra edges appear, we simply use augment_aut to add a reflexive edge for each of them at every node.

Furthermore, the gateway might be incorrect and generate some Returns that are not allowed by the source protocol. If we were to intersect the source protocol automaton with the gateway automaton, these Return edges would disappear. Thus before performing the intersection, we additionally use augment_aut to create a version of the source protocol automaton with a reflexive edge for each Return at every node.

In summary, we compute $A^{\diamond}_{P_s} = \mathsf{augment\_aut}(A_{P_s}, \mathsf{mtl\_edges})$. Then, we compute $A^{P_s}_G = A^{\diamond}_{P_s} \cap A_G$, the intersection of $A^{\diamond}_{P_s}$ with the gateway protocol automaton. The only effect of this intersection is to chain the handlers of the gateway automaton according to what is allowed by the source protocol automaton.

Checking compatibility

Finally, our goal is to check whether the restricted gateway automaton $A^{P_s}_G$ is compatible with all of the other automata. By compatible, we mean a subset: the gateway automaton should produce some of the message sequences that are allowed, but need not produce all of them, and should not produce message sequences that are not allowed. The desired relationships are as follows, where $A \bowtie B$ means that A is compatible with B:

$A^{P_s}_G \bowtie A_{P_s}$	Subsets of expected Returns
$A^{P_s}_G \bowtie A_{P_t}$	Subsets of expected Sends
$A^{P_s}_G \bowtie A_{sessions}$	Subsets of expected operation sequences
$A^{P_s}_G \bowtie A_{tcp}$	Subsets of expected operation sequences

Table 4.1: Relationships between automata

In each of the multidirectional cases (source protocol vs. gateway and gateway vs. target protocol), one direction is forced to be an equality, so it is sufficient to do a subset check in the other direction. Note that in the case of the target

protocol, this property follows directly from the fact that we have replaced all of the Receive edges by Receive(ϵ) in the verification setup phase. In the Accept case, equality on acceptation if forced by the z2z compiler. In the Receive case, if the sends are correct, the z2z compiler forces the receives to be correct.

In each case, to check compatibility of $A_G^{P_s}$ with an automaton $A_x \in \{A_{P_s}, A_{P_t}, A_{sessions}, A_{tcp}\}$, we first use filter_aut to restrict $A_G^{P_s}$ to the kinds of edges that are found in A_x. The filter ϕ_{P_s} is used for comparison with the source protocol automaton, ϕ_{P_t} for comparison with the target protocol automaton, $\phi_{session}$ for comparison with the session automaton, and ϕ_{tcp} for comparison with the TCP automaton. The resulting filtered automaton is defined on exactly the same set of edges that are used to define the automaton A_x with which it is being compared. Finally, we check that the resulting filtered automaton is a subset of A_x.

The subset check $A \subseteq B$ is implemented as $A \cap \overline{B}$, where $\overline{B}$ is the complement of B. The result of this intersection, if not empty, describes the automaton of failures. We select one of the smaller path produced by this automaton and display it to the user as a counter example, describing a rule violation.

4.7 Evaluation

To assess our approach, we have implemented the SIP/RTSP, SLP/UPnP, SMTP/HTTP and HTTP/SMTP gateways described in the case studies of Section 4.2. We have implemented our gateways on a Single Board Computer (SBC) to represent the kind of limited but inexpensive or energy-efficient devices that are found in PDAs, mobile devices and home appliances. We use a Eukréa CPUAT91 card,[3] based on a 200 MHz ARM9 processor. The SBC has 32MB of SDRAM, 8MB of flash memory, an Ethernet controller, and runs a minimal Linux 2.6.20 kernel. For the SIP/RTSP experiment, we use the open-source Linphone video-phone client[4] and an Axis RTSP camera. For the SLP/UPnP experiment, we use a handcrafted SLP client based on the INDISS framework [14] and a UPnP service provided by CyberLink.[5] For the SMTP/HTTP/SMTP experiment, we use the multi-threaded SMTP test client and server distributed with Postfix [51] to stress the generated gateways.

			Input specification (lines of z2z code)			Parser wrapper (lines of C code)	Z2z gateway (size in KB)		
			PS	MS	MT	Parser or wrapper	Generated modules	Runtime system	Total
SIP/RTSP		SIP	24	118	102	168	72	80	152
		RTSP	20	104		210			
		SDP	-	12		52			
		SDP_media	-	15		83			
SLP/UPnP		SLP	12	21	5	166	44	80	124
		SSDP	6	31		223			
		HTTP	9	43		178			
SMTP/HTTP	UDP	SMTP	10	23	83	96	40	80	120
		HTTP	9	92		103			
	TCP alive	SMTP	10	23	71	96	36		116
		HTTP	9	64		105			
	TCP non-alive	SMTP	10	23	83	96	36		
		HTTP	9	92		114			
HTTP/SMTP	UDP	HTTP	9	43	69	160	32	80	112
		SMTP	10	23		44			
	TCP alive	HTTP	9	43	63	488	36		116
		SMTP	10	23		34			
	TCP non-alive	HTTP	9	43	75	190	32		112
		SMTP	10	23		44			

Figure 4.14: The size of the input specifications and the generated gateway

Figure 4.14 shows the size of the various specifications that must be provided to generate each gateway. A protocol specification module is independent of the targeted gateway and thus can be shared by all gateways relevant to the protocol. The message translation module for the SLP/UPnP gateway is particularly simple, being only 5 lines of code. Indeed, the complete z2z specification for this gateway is less than 100 lines of code. As described in Section 4.3.3, the message specification module must provide a parser for incoming messages. For our experiments, we have implemented simple parsers for each message type, amounting to at most 488 lines of C code. Each of the generated gateways does not exceed, in the worst case, 150KB of compiled C code, including 80KB for the runtime system.

[3] Eukréa. http://www.eukrea.com/

[4] Linphone: http://www.linphone.org/

[5] CyberLink: http://www.cybergarage.org/net/upnp/java/

Native service

	SIP↔SIP	RTSP↔RTSP
Time	351	701
	SLP↔SLP	UPnP↔UPnP
Time	2	58

Z2z

	SIP/RTSP	SLP/UPnP
Time	986	78

(a)

Native service - SMTP/SMTP

Nb client	Mail size (KB)	Nb Mail	Time
1	1	1	10
1	10	1	10
1	10	10	15
10	10	10	15

Z2z - SMTP/HTTP/SMTP

	Nb client	Mail size (KB)	Nb Mail	Time	increase factor
UDP	1	1	1	50	5.0
	1	10	1	65	6.5
	1	10	10	645	43.0
	10	10	10	146	9.7
TCP alive	1	1	1	48	4.8
	1	10	1	50	5.0
	1	10	10	410	27.3
	10	10	10	98	6.5
TCP non-alive	1	1	1	53	5.3
	1	10	1	53	5.3
	1	10	10	400	26.6
	10	10	10	111	7.4

(b)

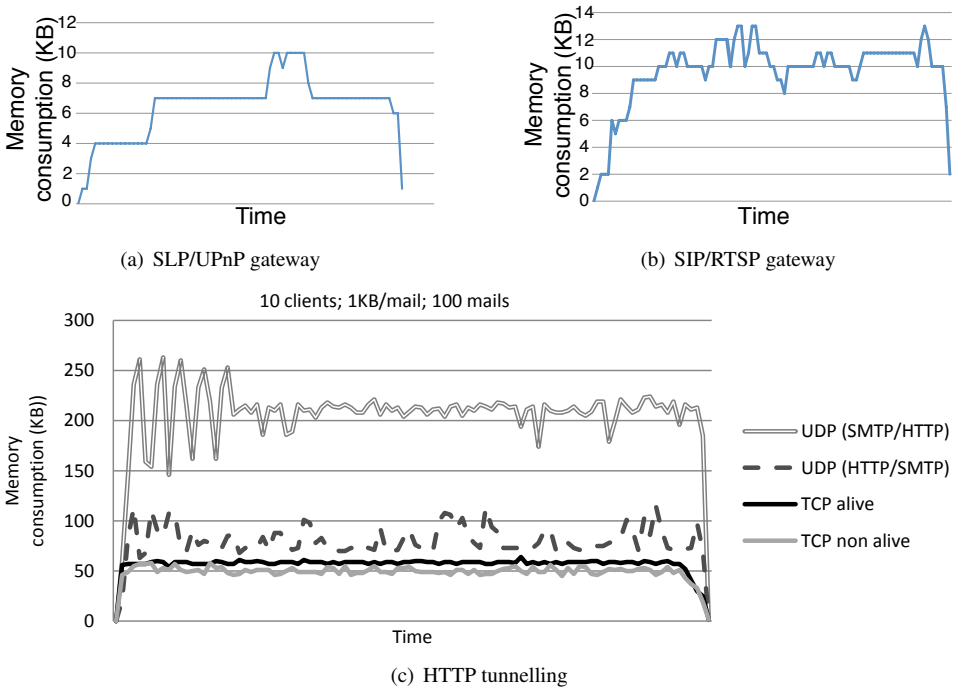
Figure 4.15: Native service vs. z2z (ms)

Figure 4.15 shows the response time for the SIP/RTSP, SLP/UPnP, and SMTP/HTTP/SMTP gateways, as well as the native protocol communication costs. In each case, at the client side we measure the time from sending an initial request to receiving a successful response. These experiments were performed using the loopback interface to remove network latency. As illustrated in Figure 4.15(a), the z2z implementation of the SIP/RTSP gateway does not introduce any overhead as compared to an ideal case of zero-cost message translation, since its response time is less than the sum of the response times for SIP and RTSP separately. Indeed, as illustrated in Figure 4.7, RTSP requests are send before the completion of the SIP `INVITE` handler leading to an interweaving of SIP and RTSP transactions. The response time for the SLP/UPnP gateway is a little more than that of SLP and UPnP separately. The cost of the z2z gateway is due in part to the polling done by the main thread in the case of asynchronous responses, as described in Section 4.5.3. This strategy introduces some time overhead, but reduces memory requirements. The response time for UPnP, furthermore, depends heavily on the stack that is used. If we use a Siemens stack[6] rather than the CyberLink stack, the native UPnP time rises to 593ms, substantially higher than the gateway cost.

Figure 4.15(b) shows the performance of the gateways generated in order to tunnel SMTP traffic into HTTP. We consider three types of tunnel according to the nature of transport layer protocol being used to exchange either asynchronously (i.e. UDP or $TCP_{non\text{-}alive}$) or synchronously (i.e. TCP_{alive}) messages between the two tunnel end-points. When sending one e-mail, with a size from 1KB up to 10 KB, passing through a HTTP tunnel roughly increases the native response time by a factor of 5, whatever tunnel is considered. This overhead comes primarily from the mailing process. According to the SMTP RFC5321, sending an e-mail involves sending at least 5 SMTP commands and acknowledgements, of at most 512 bytes each, and splitting the mail content in order to send chunks of at most 998 bytes. Commands, acknowledgements, and data chunks are packets that need to be encapsulated into the HTTP message and de-encapsulated on both sides of the HTTP tunnel. Increasing the number of packets being encapsulated at tunnel end-points inherently increases the response time. Our experimental results show that sending a sequence of 10 emails increases the response times at most by 43 as compared to a native implementation in the worst case. However, when the mail is sent in parallel (10 clients), response times only increase by a factor of at most 10. The latter result highlights the efficiency of the parallelism provided by our generated gateways. Furthermore, note that the SMTP test server simply throws away the messages received from network without processing them, whereas the tunnel end-points must process them, therefore increasing the response time. The native response time obtained with a real deployed SMTP server such as Postfix [51] may be up to 18x slower than the SMTP test server, which is much closer to the response times obtained via the generated gateways.

Finally, Figure 4.16 shows the amount of dynamic memory used during the lifetime of each of our generated gateways. The memory footprint is directly related to the network input/output traffic. When idle, the memory consumption is low and does not exceed 2KB for the considered gateways. At peak time, both the SIP/RTSP and SLP/UPnP gateways consume at most 14KB, as shown in Figs. 4.16(a) and 4.16(b). Comparatively, the SMTP/HTTP/SMTP gateways based on TCP use only at most 60KB to process one hundred KB mails sent by 10 simultaneous clients. However, the

[6] Siemens UPnP: http://www.plug-n-play-technologies.com/



(a) SLP/UPnP gateway

(b) SIP/RTSP gateway

(c) HTTP tunnelling

Figure 4.16: Dynamic memory consumption (KB)

memory consumption of the UDP-based gateway may reach up to 159KB-260KB, as shown in Figure 4.16(c). This overhead is inherent in the use of UDP, as a dedicated 1500 byte buffer is allocated for each incoming UDP packet. TCP, on the other hand, enables reading variable-size messages into a stream.

4.8 Related Work

Other approaches to interoperability. The middlewares ReMMoC [47], RUNES [33], MUSDAC [86], and BASE [6] for use in networked devices allow the device code to be developed independently of the underlying protocol. Plug-ins then select the most appropriate communication protocol according to the context. Many applications, however, have not been developed using such middleware systems and cannot be modified because their source code is not available.

Bridges provide interoperability without code modification. Direct bridges, such as RMI-IIOP[7] and IIOP-.NET,[8] provide interoperability between two fixed protocols. A direct bridge must thus be developed separately for every pair of protocols between which interaction is needed. The diversity of protocols that are used in a networked home implies that this is a substantial development task. Indirect bridges such as Enterprise Service Buses (ESBs) [25], translate messages to and from a single fixed intermediary protocol. This approach reduces the development effort, but may limit expressiveness, as some aspects of the relevant protocols may not be compatible with the chosen intermediary protocol. INDISS [14] and NEMESYS [11] also use a single intermediary protocol, but one that is specific to the protocols between which interoperability is desired. Still, none of these approaches addresses the problem of implementing the bridge, which requires a thorough knowledge of the protocols involved and low-level network programming. This makes it difficult to quickly integrate devices that use an unsupported protocol into a home environment.

Z2z can be used in the context of either direct or indirect bridges. Our approach targets the weak point of both: the difficulty of bridge development. We propose a high-level interface definition language that abstracts away from network details, makes relevant protocol properties explicit, provides static verification at the specification level, and automatically generates low-level code.

[7] RMI-IIOP:http://java.sun.com/products/rmi-iiop/
[8] IIOP-.NET: http://iiop-net.sourceforge.net/index.html

Compilation. Z2z uses a number of advanced compilation techniques to be able to provide a high-level notation while still generating safe and efficient code. Krishnamurthi *et al.* [55] pioneered the use of continuations to overcome the asynchrony common in web programming. Our implementation of continuations in C code is based on that presented by Friedman *et al.* [45]. Our dataflow analysis uses standard techniques [3], adapted to the operations of the z2z DSL. Finally, reference counting has long been used to replace garbage collection [26].

4.9 Conclusion

In this chapter, we have presented a generative language-based approach, z2z to simplify gateway construction, a problem that has not been considered by previous frameworks for gateway development. Z2z is supported by a runtime system that hides low-level networking intricacies and a compiler that checks essential correctness properties and produces efficient code. We have used z2z to automatically generate gateways between SIP and RTSP, between SLP and UPnP, and between SMTP and SMTP via HTTP. The gateway specifications are 100-400 lines of z2z code while the generated gateways are at most 150KB of compiled C code and run with a runtime memory footprint of less than 260KB, with essentially no runtime overhead.

Chapter 5

Conclusion

This chapter concludes the thesis. We start by a brief summary of the contributions presented in the document and then give some directions for future work.

5.1 Summary of contributions

In this thesis, we have shown that domain-specific languages (DSLs) can successfully reduce the level of expertise required to build efficient and robust distributed systems, putting service programming within the reach of average developers. We have presented three contributions in this area.

The first contribution is a DSL-based approach for creating telephony services based on the SIP protocol. We have defined a domain-specific virtual machine for SIP, providing the programmer with a high-level interface dedicated to telephony service development. Additionally, we have introduced a DSL named SPL that offers high-level notations and abstractions for service development. A variety of services have been written in SPL, demonstrating the usability and ease of programming of the language. Its robustness has been a key factor in expediting service deployment.

The second contribution is Zebu, a DSL-based approach for the development of network application protocol-handling layers. We have demonstrated, through various experiments, that this approach is a reliable alternative to manual development. Zebu-generated code has good performance and has a significantly lower memory footprint than comparable existing manually encoded solutions, while guaranteeing robustness and performance.

The third contribution is z2z, a generative approach to gateway construction that enables communication between devices that use incompatible protocols. Z2z includes a compiler that checks essential correctness properties, and a runtime system that hides low-level details from the gateway programmer. We have used z2z to automatically generate gateways between various incompatible protocols. The generated gateways run with a low runtime memory footprint, and with essentially no runtime overhead.

5.2 Future Work

In the remainder of this section, we present some directions for future work, some of which is already in progress.

A reconfigurable multi-protocol and multi-radio gateway Recent years have seen a phenomenal growth in various networking devices ranging from sensors to mobile devices. This has been combined with the advances in wireless networking technologies such as Bluetooth, Radio-Frequency Identification (RFID), Ultra-wideband (UWB), and WiFi. These changes are promoting the deployment of a worldwide Internet of Things (IoT). Any computing units that embed short range mobile transceivers should able to interact directly with each other at anytime, anywhere through an always-on connectivity. In this context, the Internet can be viewed as one backbone network that interconnects a huge collection of small separate networks. Examples of such separate networks include sensor networks, vehicular networks, MANETs, and internal networks of corporations.

Unfortunately, however, the devices used in these domains are highly heterogeneous in terms of both hardware and software. Traditionally, because of substantial resource constraints (battery, memory, CPU, etc.), these devices provide only a limited set of wireless networking technologies and support few application protocols. As described in Chapter 4, z2z addresses the protocol incompatibility issue by automatically generating gateways. Nevertheless, z2z-based gateways rely on the wireless networking capabilities of host devices, preventing them from communicating with devices that use unsupported technologies. Over the last decade, software defined radio (SDR) has gained an increasing interest, as new radio standards are deployed without substantially supplanting existing ones [35]. With the unique combination of size and power efficiency of reconfigurable hardware platforms such as FPGAs, SDR becomes an appealing approach. SDR is easy to reconfigure since most of the key components in the communication system, including the physical layer, are pushed into software. In addition, FPGAs are increasingly used in embedded devices to provide good performance while consuming low resources.

We plan to extend z2z by enabling gateways deployed in appliances with constrained resources such as telephones or set top boxes to be dynamically updated according to the devices currently present in their environment. We are currently investigating the use of FPGAs to provide software defined radio capabilities and a fine grained usage of the limited appliance resources. We will also investigate the design of a reconfigurable communication middleware to provide secure and robust dynamic reconfiguration capabilities to the underlying reconfigurable hardware platform.

Efficient event-based multicore programming for system services The advent of multicore hardware is today a reality. Nevertheless, current systems and applications are unable to fully exploit these new architectures: performance is stalling, even though the number of cores is rising. This inability to exploit the available processing power is inhibiting the development of innovative, demanding software services. This is particularly true for operating system kernels and system services, which are known to be complex to develop and optimize. In fact, efficiently exploiting multicore architectures is one of the main research and industrial challenges of the coming years.

One of the main reasons why current system software does not take full advantage of multicore architectures is that it is mostly designed using the traditional, thread-based programming model. System services that were developed using the thread-based model and that were achieving very good performance on monocore architectures, often fail to yield good performance on multicore architectures. The inter-thread synchronization code used to manage concurrent accesses to shared data structures is very complex and error-prone [39, 44, 66, 67, 68, 80]: traditional locking-based techniques can induce bugs such as priority inversions, deadlocks, or even livelocks. To avoid these problems, developers often design programs that use coarse grain synchronization, which limits parallelism. While on monocore architectures, this limit on parallelism is often hidden due to the low pressure of coarse-grained, pseudo-parallel executions, this is no longer the case on multicore architectures. Due to the use of coarse-grained parallelism, services do not fully exploit all the cores available.

Event-based programming is a promising approach for parallel programming as it significantly reduces the need to write complex synchronization code. Most event-driven runtimes, however, have been designed for monocore architectures. The challenge in a multi-core setting is to provide atomic execution of event handlers, free from interference with event handlers running on other cores. Event coloring, in which event handlers are annotated to indicate bad interactions with other handlers, is a promising approach to address this problem, but has not yet been successfully implemented in an efficient manner.

We plan to introduce optimizations into an event-coloring event-based runtime, to make it possible for applications to scale linearly with the number of cores. Optimizations to be considered include improving the use of caches, to reduce the usage of synchronization code. Our goal is to propose a DSL that will be used for programming system services and will drive the optimization of the event-based runtime. To the best of our knowledge, there exist very few DSLs related to this context [57, 23].

DSLs as a software architecture for future system services According to Moore's Law, the number of transistors on a chip roughly doubles every two years. This prediction has proved to be uncannily accurate, and is expected to continue for at least several years. To take full advantage of the gains allowed by this trend, processor makers favor multi-core chip designs in almost all application domains, from high performance computers to embedded systems. Indeed, some vendors have already announce smartphones with quad core processors. Similarly, numerous other hardware components (*i.e.*, memory, screen, network) are expected to provide in the near future capabilities well beyond what they do today.

This phenomenal growth of various device technologies is promoting the deployment of advanced end user services whose needs are difficult to anticipate. However, the development of these services will require that the service programmer have extensive expertise in many fields. As demonstrated by the contributions presented in this thesis, we

argue that a DSL-based software architecture is a candidate of choice for the development of system services of the future.

As system services are becoming more and more complicated to develop, they may require to use several DSLs, each being designed for a particular subproblem. However, the proliferation of different languages may increase the difficulty of their integration in a complex system and thus limit the practical benefits of DSLs. We think that one of the most challenging directions for future work is to address this issue of DSL composition for system services. DSL composition has been already investigated [72, 90], mainly in the context of embedded DSLs. However, to the best of our knowledge, there exist very few approaches for composing DSLs in the context of system services. Based on our experience in developing DSLs for distributed systems, we plan to investigate this research area.

Bibliography

[1] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005.

[2] H. Agrawal, R. DeMillo, R. Hathaway, Wm. Hsu, W. Hsu, E. Krauser, R.J. Martin, A. Mathur, and E. Spafford. Design of mutant operators for the C programming language. Technical Report SERC-TR-41-P, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana, 1989.

[3] Andrew Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

[4] B.R.T. Arnold, A. van Deursen, and M. Res. An algebraic specification of a language describing financial products. In *IEEE Workshop on Formal Methods Application in Software Engineering*, pages 6–13, April 1995.

[5] Jeroen Arnoldus, Jeanot Bijpost, and Mark van den Brand. Repleo: a syntax-safe template engine. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 25–32, NY, USA, 2007. ACM.

[6] Christian Becker, Gregor Schiele, Holger Gubbels, and Kurt Rothermel. Base: A micro-broker-based middleware for pervasive computing. In *PERCOM '03: Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, page 443, Washington, DC, USA, 2003. IEEE Computer Society.

[7] Tegawendé F. Bissyandé, Laurent Réveillère, Yérom-David Bromberg, Julia L. Lawall, and Gilles Muller. Bridging the gap between legacy services and web services. In Indranil Gupta and Cecilia Mascolo, editors, *Middleware*, volume 6452 of *Lecture Notes in Computer Science*, pages 273–292. Springer, 2010.

[8] E. Bjarnason. Applab: a laboratory for application languages.

[9] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice & Experience*, 18(9):807–820, September 1988.

[10] Nikita Borisov, David J. Brumley, Helen J. Wang, John Dunagan, Pallavi Joshi, and Chuanxiong Guo. A generic application-level protocol analyzer and its language. In *14th Annual Network & Distributed System Security Symposium*, San Diego, CA, USA, February 2007.

[11] Yérom-David Bromberg. *Solutions to middleware heterogeneity in open networked environment*. Phd Thesis, INRIA/UVSQ, 2006.

[12] Yérom-David Bromberg, Paul Grace, and Laurent Réveillère. Starlink: Runtime interoperability between heterogeneous middleware protocols. In *ICDCS*, pages 446–455. IEEE Computer Society, 2011.

[13] Yérom-David Bromberg, Paul Grace, Laurent Réveillère, and Gordon Blair. Bridging the interoperability gap: Overcoming combined application and middleware heterogeneity. In *ACM/IFIP/USENIX International Middleware Conference*, dec 2011. To appear.

[14] Yérom-David Bromberg and Valérie Issarny. INDISS: Interoperable discovery system for networked services. In *Proceedings of the 6th International Middleware Conference*, pages 164–183, Grenoble, France, November 2005.

[15] Yerom-David Bromberg, Laurent Réveillère, Julia Lawall, and Gilles Muller. Automatic generation of network protocol gateways. In *ACM/IFIP/USENIX 10th International Middleware Conference*, pages 1–20, Urbana Champaign, IL, USA, November 2009.

[16] L. Burgy, L. Caillot, C. Consel, F. Latry, and L. Réveillère. A comparative study of SIP programming interfaces. In *Proceedings of the ninth International Conference on Intelligence in service delivery Networks (ICIN 2004)*, Bordeaux, France, October 2004.

[17] L. Burgy, C. Consel, F. Latry, J. Lawall, L. Réveillère, and N. Palix. Language technology for internet-telephony service creation. In *IEEE International Conference on Communications*, 2006.

[18] L. Burgy, C. Consel, F. Latry, N. Palix, and L. Réveillère. A high-level, open ended architecture for sip-based services. In *Proceedings of the tenth International Conference on Intelligence in service delivery Networks (ICIN 2006)*, pages 364–365, Bordeaux, France, May 2006.

[19] L. Burgy, C. Consel, F. Latry, N. Palix, and L. Réveillère. Routing device for an ip telephony system, feb 2008. European Patent EP1887774.

[20] Laurent Burgy. *Approche langage au développement du support protocolaire d'applications réseaux*. These, Université Sciences et Technologies - Bordeaux I, April 2008.

[21] Laurent Burgy, Laurent Réveillère, Julia L. Lawall, and Gilles Muller. A language-based approach for improving the robustness of network application protocol implementations. In *26th IEEE International Symposium on Reliable Distributed Systems*, pages 149–158, Beijing, October 2007.

[22] Laurent Burgy, Laurent Réveillère, Julia L. Lawall, and Gilles Muller. Zebu: A language-based approach for network protocol message processing. *IEEE Trans. Software Eng.*, 37(4):575–591, 2011.

[23] Brendan Burns, Kevin Grimaldi, Alexander Kostadinov, Emery D. Berger, and Mark D. Corner. Flux: A language for programming high-performance servers. In *USENIX Annual Technical Conference*, Boston, MA, USA, May 2006. USENIX Association.

[24] S. Chandra and J. Larus. Experience with a language for writing coherence protocols. In *Proceedings of the 1st USENIX Conference on Domain-Specific Languages*, Santa Barbara, CA, October 1997.

[25] David Chappell. *Enterprise Service Bus*. O'Reilly, 2004.

[26] Jacques Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.

[27] C. Consel, H. Hamdi, L. Réveillère, L. Singaravelu, H. Yu, and C. Pu. Spidle: A DSL approach to specifying streaming application. In *Second International Conference on Generative Programming and Component Engineering*, Erfurt, Germany, September 2003.

[28] C. Consel, F. Latry, L. Réveillère, and P. Cointe. A generative programming approach to developing dsl compilers. In R. Gluck and M. Lowry, editors, *Fourth International Conference on Generative Programming and Component Engineering (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 29–46, Tallinn, Estonia, September 2005. Springer-Verlag.

[29] C. Consel and L. Réveillère. *Domain-Specific Program Generation; International Seminar, Dagstuhl Castle*, chapter A DSL Paradigm for Domains of Services: A Study of Communication Services, pages 165 – 179. Number 3016 in Lecture Notes in Computer Science, State-of-the-Art Survey. Springer-Verlag, 2004.

[30] Steve Cook, Gareth Jones, Stuart Kent, and Alan Wills. *Domain-specific development with visual studio dsl tools*. Addison-Wesley Professional, first edition, 2007.

[31] M. Cortes and J. R. Ensor. Narnia: A virtual machine for multimedia communication services. In *Proceedings of the Fourth International Symposium on Multimedia Software Engineering*, pages 246 – 254, 2002.

[32] M. Cortes, J.R. Ensor, and J.O. Esteban. On SIP performance. Technical report, Bell Labs Technical Journal 3, 2004.

[33] Paolo Costa, Geoff Coulson, Cecilia Mascolo, Luca Mottola, Gian Pietro Picco, and Stefanos Zachariadis. Reconfigurable component-based middleware for networked embedded systems. *In International Journal of Wireless Information Networks*, 14(2):149–162, June 2007.

[34] D. Crocker and P. Overell. Augmented BNF for syntax specifications: ABNF. Internet Engineering Task Force: RFC 2234, 1997.

[35] M. Cummings and S. Haruyama. Fpga in the software radio. *Communications Magazine, IEEE*, 37(2):108 –112, feb 1999.

[36] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[37] J. Deruelle, M. Ranganathan, and D. Montgomery. Programmable active services for JAIN SIP. Technical report, National Institute of Standards and Technology, June 2004.

[38] Conal Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In *Proceedings of the 1st USENIX Conference on Domain-Specific Languages*, Santa Barbara, CA, October 1997.

[39] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware Java runtime. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–255, San Diego, California, USA, 2007. ACM.

[40] Exmap-console. http://projects.o-hand.com/exmap-console.

[41] The eXtended osip library. http://savannah.nongnu.org/projects/exosip/, 2003.

[42] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[43] K. Fisher and R. Gruber. PADS: a domain-specific language for processing ad hoc data. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 295–304. ACM, 2005.

[44] Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–133, Dublin, Ireland, 2009. ACM.

[45] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press, 1992.

[46] Yaron Y. Goland, Ting Cai, Paul Leach, and Ye Gu. Simple service discovery protocol/1.0: Operating without an arbiter. http://quimby.gnus.org/internet-drafts/draft-cai-ssdp-v1-03.txt, October 1999.

[47] Paul Grace, Gordon S. Blair, and Sam Samuel. A reflective framework for discovery and interaction in heterogeneous mobile environments. *SIGMOBILE Mob. Comput. Commun. Rev.*, 9(1):2–14, 2005.

[48] J. Greenfield, K. Short, S. Cook, S. Kent, and J. Crupi. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, August 2004.

[49] N.K. Gupta, L. J. Jagadeesan, E. E. Koutsofios, and D. M. Weiss. Auditdraw: Generating audits the fast way. In *Proceedings of the Third IEEE Symposium on Requirements Engineering*, pages 188–197, January 1997.

[50] P. Hazel. PCRE - perl compatible regular expressions. http://www.pcre.org/, 2006.

[51] Ralf Hildebrandt and Patrick Koetter. *The book of Postfix: state-of-the-art message transport*. NO-STARCH, 2005. http://www.postfix.org/.

[52] iptel.org. *SER Developer's guide*, September 2003.

[53] Java Community Process. *SIP Servlet API*, 2003. http://jcp.org/en/jsr/detail?id=116.

[54] S. C. Johnson. Yacc: Yet another compiler compiler. Technical report, Bell Telephone Laboratories, 1975.

[55] Shriram Krishnamurthi, Peter Walton Hopkins, Jay McCarthy, Paul T. Graunke, Greg Pettyjohn, and Matthias Felleisen. Implementation and use of the PLT Scheme web server. *Higher-Order and Symbolic Computation*, 20(4):431–460, 2007.

[56] S. Krishnamurthy. TinySIP: Providing seamless access to sensor-based services. In *3rd International Conference on Mobile and Ubiquitous Systems: Networking and Services*, number 4611 in Lecture Notes in Computer Science, pages 1–9, 2006.

[57] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *Proceedings of the 2007 USENIX Annual Technical Conference*, pages 87–100, Santa Clara, CA, USA, June 2007. USENIX Association.

[58] J. Kuthan. Sip express router (SER). *IEEE Network Magazine*, July 2003.

[59] Fabien Latry. *Approche langage au développement logiciel : application au domaine des services de téléphonie sur IP.* These, Université Sciences et Technologies - Bordeaux I, sep 2007.

[60] S. Leggio, J. Manner, A. Hulkkonen, and K. Raatikainen. Session initiation protocol deployment in ad-hoc networks: A decentralized approach. In *2nd International Workshop on Wireless Ad-hoc Networks*, 2005.

[61] J. Lennox and H. Schulzrinne. CPL: A language for user control of internet telephony services. Internet Engineering Task Force, IPTEL WG, November 2000.

[62] L. J. Lennox. *Services for Internet Telephony*. PhD thesis, Columbia University, December 2003.

[63] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. Technical report, Bell Telephone Laboratories, 1975.

[64] Linphone. An open-source SIP video-phone for Linux and Windows. http://www.linphone.org/.

[65] Livemedia. Streaming media. http://www.livemediacast.net/about/library.cfm.

[66] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP '07: Proceedings of 21st ACM SIGOPS symposium on Operating systems principles*, pages 103–116, Stevenson, Washington, USA, 2007. ACM.

[67] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–339, Seattle, WA, USA, 2008. ACM.

[68] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-aid: Detecting and surviving atomicity violations. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 277–288, Washington, DC, USA, 2008. IEEE Computer Society.

[69] A. Madhavapeddy. *Creating High-Performance, Statically Type-Safe Network Applications*. PhD thesis, Cambridge University, 2007.

[70] P. J. McCann and S. Chandra. PacketTypes: Abstract specifications of network protocol messages. In *ACM SIGCOMM 2000 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 321–333, 2000.

[71] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: An IDL for hardware programming. In *4th Symposium on Operating System Design and Implementation (OSDI 2000)*, pages 17–30, San Diego, CA, USA, October 2000.

[72] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37:316–344, December 2005.

[73] Microsoft. *Live Communications Server Application API*, 2005.

[74] Sun Microsystems. The JAIN SIP API specification v1.1. Technical report, Sun Microsystems, June 2003.

[75] Moizard. The GNU oSIP library. http://www.gnu.org/software/osip, June 2001.

[76] Masahide Nakamura, Pattara Leelaprute, K. Matsumoto, and Tohru Kikuno. On detecting feature interactions in the programmable service environment of Internet telephony. *Computer Networks (Amsterdam, Netherlands: 1999)*, 45(5):605–624, August 2004.

[77] Nicolas Palix. *Langages dédiés au développement de services de communications*. These, Université Sciences et Technologies - Bordeaux I, September 2008.

[78] Nicolas Palix, Charles Consel, Laurent Réveillère, and Julia Lawall. A stepwise approach to developing languages for SIP telephony service creation. In *IPTComm '07: Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications*, pages 79–88, New York, NY, USA, 2007. ACM.

[79] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: A yacc for writing application protocol parsers. In *Proceedings of the 6th ACM SIGCOMM on Internet measurement*, pages 289–300, 2006.

[80] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS '09: Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–36, Washington, DC, USA, 2009. ACM.

[81] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[82] Colin Perkins. *RTP - Audio and Video for the Internet*. Addison-Wesley, 2003.

[83] J. Postel. RFC 821: Simple mail transfer protocol, 1982.

[84] C. Pu, A. Black, C. Cowan, J. Walpole, and C. Consel. Microlanguages for operating system specialization. In *1st ACM-SIGPLAN Workshop on Domain-Specific Languages*, Paris, France, January 1997. Computer Science Technical Report, University of Illinois at Urbana-Champaign.

[85] X. Qie, R. Pang, and L. Peterson. Defensive programming: Using an annotation toolkit to build DoS-resistant software. In *5th Symposium on Operating System Design and Implementation (OSDI 2002)*, pages 45–60, 2002.

[86] Pierre-Guillaume Raverdy, Valerie Issarny, Rafik Chibout, and Agnes de La Chapelle. A multi-protocol approach to service discovery and access in pervasive environments. In *The 3rd Annual International Conference on Mobile and Ubiquitous Systems: Networks and Services*, pages 1–9, San Jose, CA, USA, July 2006.

[87] Nokia research. Sofia-SIP. http://opensource.nokia.com/projects/sofia-sip/, 2006.

[88] A. B. Roach. Session Initiation Protocol (SIP)-Specific Event Notification. Technical Report 3265, IETF, June 2002.

[89] J. et al. Rosenberg. SIP : Session Initiation Protocol. Technical Report 3261, IETF, June 2002.

[90] Jesus Sanchez Cuadrado and Jesus Garcia Molina. A model-based approach to families of embedded domain-specific languages. *IEEE Transactions on Software Engineering*, 35:825–840, November 2009.

[91] R. Sparks, A. Hawrylyshen, A. Johnston, J. Rosenberg, and H. Schulzrinne. Session initiation protocol (SIP) torture test messages. Internet Engineering Task Force: RFC 4475, 2006.

[92] A. Spencer. Asterisk: The open source PBX. http://www.asterisk.org.

[93] Guy L. Steele Jr. Lambda, the ultimate declarative. Technical Report 379, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, November 1976.

[94] P. Stuedi, M. Bihr, A. Remund, and G. Alonso. SIPHoc: Efficient SIP middleware for ad hoc networks. In *Proceedings of the 8th ACM/IFIP/USENIX International Conference on Middleware*, 2007.

[95] S. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. In *17th IEEE Symposium on Reliable Distributed Systems*, pages 135–143, West Lafayette, IN, October 1998.

[96] The videolan project. http://www.videolan.org/vlc/.

[97] W3C. *CCXML: "Voice browser call control: CCXML version 1.0."*. http://www.w3.org/TR/ccxml/.

[98] S. Wanke, M. Scharf, S. Kiesel, and S. Wahl. Measurement of the SIP parsing performance in the SIP Express Router. In *Dependable and Adaptable Networks and Services*, number 4606 in Lecture Notes in Computer Science, pages 103–110, 2007.

[99] Duane Wessels. *Squid: The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004.

[100] X. Wu and H. Schulzrinne. Programmable end system services using sip. In *Proceedings of The IEEE International Conference on Communications 2002*. IEEE, 2003.

[101] Xiaotao Wu and Henning Schulzrinne. Handling feature interactions in the language for end system services. In Stephan Reiff-Marganiec and Mark Ryan, editors, *FIW*, pages 270–287. IOS Press, 2005.

[102] Yiqun Xu, Luigi Logrippo, and Jacques Sincennes. Detecting feature interactions in CPL. *Journal of Network and Computer Applications*, December 2005.

Appendix A

The SPL Language

This chapter presents the syntax of the SPL language using a BNF-like notation. `typewriter font` is used for keywords. *italic font* is used for nonterminals. Parentheses are used for grouping. A superscript $*$ denotes 0 or more occurrences of the preceding item. A superscript $+$ denotes 1 or more occurrences of the preceding item. A superscript ? indicates that the preceding item is optional. Identifiers (*ident*) and various other base type terms (*char-val*, *num-val*, etc.) are left unspecified.

A.0.1 Lexical Conventions

SPL Comments Comments are C-like comments. They start with the characters `/*` and end with the characters `*/`. C++ comments style can also be used; all characters from the two characters `//` until the end of the line are considered as part of a comment. Comments are treated as blanks.

Identifiers. An identifier is a sequence of letters, digits and _ (the underscore character) starting with a letter or an underscore. A letter can be any of the 52 lowercase and uppercase letters from the ASCII character set. The current implementation places no limit on the number of characters in an identifier.

ident ::= (*letter* | _) (*letter* | *digit* | _)*
letter ::= `A..Z` | `a..z`
digit ::= `0..9`

There are several kinds of identifiers :

- *ident* which defines identifiers (for variables, functions, branches and service name)
- *headerId* which defines SIP headers in the RFC fashion, with a colon at the end of the header name
- *eventIdent* which defines SIP events in the RFC fashion, as an alphanumeric sequence of characters with possibly some '.' inside when template event packages are used
- *uriScheme* which is defined as *scheme* in RFC 3261

Header Names A header name is a sequence of letters, digits and − (the minus character) starting with a letter and ending with a colon. A letter can be any of the 52 lowercase and uppercase letters from the ASCII character set. The current implementation places no limit on the number of characters of a header name.

headerId ::= #*letter* (*letter* | *digit* | _ | ! | % | * | - | + | ' | ` | ~)* :

Event Identifiers An event identifier is a sequence of letters, digits and special characters as defined in RFC 3265. A letter can be any of the 52 lowercase and uppercase letters from the ASCII character set. The current implementation places no limit on the number of characters of an event identifier.

eventIdent ::= *event-package* (`.` *event-template*)*
event-package ::= *token-nodot*
event-template ::= *token-nodot*
token-nodot ::= (*letter* | *digit* | `_` | `!` | `%` | `*` | `-` | `+` | `'` | `` ` `` | `~`)+

Integer Literals An integer literal is a sequence of one or more digits. Integer literals are in decimal (radix 10).

integer ::= `0` | (`1..9` *digit**)

Prefix and Infix Operator The following tokens are the SPL operators:

```
=    +    -      *         /    &&
==   !=   <      >         >=   ||
<=   !    match  notmatch
```

Keywords The identifiers below are reserved keywords:

```
BODY          bool          branch        break         bye
cancel        case          continue      default       deploy
dialog        else          event         extern        false
FIFO          foreach       forward       http          https
if            in            incoming      int           LIFO
mailto        match         notmatch      outgoing      parallel
pop           processing    push          reason        registration
request       requestURI    response      return        select
service       sip           sips          string        this
time          true          type          undeploy      uninvite
unregister    unsubscribe   uri           void          when
with
```

The following character sequences are also reserved.

```
{   }   [   ]   (   )   <   >   |
=   :   ,   ;   .
```

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.1 SPL Program

A SPL program consists of a service name and a *service* construct. When a method declaration does not specify an explicit *direction*, it applies to both incoming and outgoing. SIP method names, as defined by the rule *SIPmethod* are treated as SPL keywords.

```
program     ::= service ident {service}
service     ::= processing {declarations? session*}
              | declarations? session*
session     ::= registration {declarations? session*}
              | dialog {declarations? method+}
              | event eventIdent {declarations? method+}
              | method
method      ::= typExp direction? methodName (arg?) {stmt+}
              | typExp direction? methodName (arg?) {branch+}
branch      ::= branch ident (| ident)* {stmt+}
              | branch default {stmt+}
direction   ::= incoming
              | outgoing
methodName  ::= SIPmethod
              | ctrlMethod
SIPmethod   ::= INVITE | ACK | BYE | CANCEL | OPTIONS | MESSAGE
              | NOTIFY | PUBLISH | REGISTER | SUBSCRIBE
              | REINVITE | REREGISTER | RESUBSCRIBE
ctrlMethod  ::= deploy | undeploy
              | uninvite | unregister | unsubscribe
```

A.2 Type Expressions

SPL type expressions are defined as follows.

```
typExp     ::= simpleType
             | modifier? simpleType<integer>
             | ident
simpleType ::= void | bool | int | request | response
             | string | time | uri
modifier   ::= LIFO | FIFO
```

A.3 Function Call

A function call is defined as follows.

```
functionCall ::= ident (expList?)
expList      ::= exp (, exp)*
```

A.4 Known URI Kinds

Standard URI kinds are treated as reserved keywords. Otherwise, the URI scheme is defined according to RFC 3261.

```
uriKind   ::= http | https | mailto | sip | sips
            | uriScheme
uriScheme ::= letter (letter | digit | - | + | .)*
```

A.5 Declarations

Declarations are defined as follows.

```
declarations ::= declarations declaration
               | declaration
declaration  ::= typExp ident (= exp)? ;
               | extern typExp ident (args?) ;
               | typExp ident (args?) {stmt+}
               | type ident { (typExp ident;)+ } ;
args         ::= arg (, arg)*
arg          ::= typExp ident
```

A.6 Statements

Statements are defined as follows.

```
compound     ::= stmt
               | {stmt+}
stmt         ::= place = exp;
               | declaration
               | return exp? (branch ident)? ;
               | if(exp) compound
               | if(exp) compound else compound
               | when exp (when-headers) compound
               | when exp (when-headers) compound else compound
               | exp with {messageField (, messageField)*}
               | foreach(ident in exp) compound
               | select(exp) { selectCase* selectDefault? }
               | functionCall;
               | continue;
               | break;
               | push place exp;
when-headers ::= when-header (, when-header)*
when-header  ::= headerId typExp ident (<- const)?
selectCase   ::= case orList: (stmt)*
selectDefault ::= default: (stmt)*
orList       ::= const (| const)*
place        ::= ident(.ident)*
               | sipHeader
```

A.7 Message Modification

The `with` construct allows modification of the field values of a request or response. The `this` constant refers to the the current processed request.

```
messageField ::= reason=exp
               | headerId exp
```

A.8 Expressions

Expressions are defined as follows.

```
exp       ::= ident( . ident)*
            | constant
            | sipHeader
            | unop exp
            | exp binop exp
            | (parallel)? forward (exp)?
            | with
            | (exp)
            | reason
            | BODY
            | requestURI
            | this
            | pop place
            | functionCall
unop      ::= ! | -
binop     ::= + | - | * | / | < | > | == | != | <= | >= | && | ||
            | match | notmatch
constant  ::= true | false
            | integer | "string" | uri | sequence | response
sequence  ::= <constant (, constant)*>
            | <>
uri       ::= 'uriKind:uriString'
```

A.9 SIP Headers

SIP headers defined as mandatory in at least one method according to RFC 3261 and RFC 3265 are treated as SPL keywords.

```
sipHeader ::= CALL_ID | CONTACT | CSEQ | EVENT | FROM | MAX_FORWARDS
            | SUBSCRIPTION_STATE | TO | VIA
```

A.10 Constant Responses

Responses are defined according to RFC 3261 and RFC 3265 and are treated as SPL keywords.

```
response        ::= /ERROR responseError?
                  | /SUCCESS responseSuccess?
responseSuccess ::= /OK
                  | /ACCEPTED
  responseError ::= /CLIENT clientErr?
                  | /GLOBAL globalErr?
                  | /REDIRECTION redirectionErr?
                  | /SERVER serverErr?
redirectionErr  ::= /ALTERNATIVE_SERVICE
                  | /MOVED_PERMANENTLY
                  | /MOVED_TEMPORARILY
                  | /MULTIPLE_CHOICES
                  | /USE_PROXY
    clientErr     ::= /ADDRESS_INCOMPLETE
                  | /AMBIGUOUS
                  | /BAD_EXTENSION
                  | /BAD_REQUEST
                  | /BUSY_HERE
                  | /CALL_OR_TRANSACTION_DOES_NOT_EXIST
                  | /EXTENSION_REQUIRED
                  | /FORBIDDEN
                  | /GONE
                  | /INTERVAL_TOO_BRIEF
                  | /LOOP_DETECTED
                  | /METHOD_NOT_ALLOWED
                  | /NOT_ACCEPTABLE_HERE
                  | /NOT_ACCEPTABLE
                  | /NOT_FOUND
                  | /PAYMENT_REQUIRED
                  | /PROXY_AUTHENTICATION_REQUIRED
                  | /REQUESTURI_TOO_LONG
                  | /REQUEST_ENTITY_TOO_LARGE
                  | /REQUEST_PENDING
                  | /REQUEST_TERMINATED
                  | /REQUEST_TIMEOUT
                  | /TEMPORARILY_UNAVAILABLE
                  | /TOO_MANY_HOPS
                  | /UNAUTHORIZED
                  | /UNDECIPHERABLE
                  | /UNSUPPORTED_MEDIA_TYPE
                  | /UNSUPPORTED_URI_SCHEME
    serverErr     ::= /BAD_GATEWAY
                  | /MESSAGE_TOO_LARGE
                  | /NOT_IMPLEMENTED
                  | /SERVER_INTERNAL_ERROR
                  | /SERVER_TIMEOUT
                  | /SERVICE_UNAVAILABLE
                  | /VERSION_NOT_SUPPORTED
globalErr       ::= /BUSY_EVERYWHERE
                  | /DECLINE
                  | /DOES_NOT_EXIST_ANYWHERE
                  | /NOT_ACCEPTABLE
```

Appendix B

The Zebu Language

The syntax of the Zebu language is given in Figures B.1 to B.2 using a BNF-like notation. `typewriter font` is used for keywords. *italic font* is used for nonterminals. Parentheses are used for grouping. A superscript $*$ denotes 0 or more occurrences of the preceding item. A superscript $+$ denotes 1 or more occurrences of the preceding item. A superscript ? indicates that the preceding item is optional. Identifiers (*ident*) and various other base type terms (*char-val*, *num-val*, etc.) are left unspecified. Comments, as in ABNF, begin with `;` and cannot appear within an element, but only at the end of a line.

B.1 Protocol-Handling Layer Specification

Figure B.1 describes the BNF syntax of a message specification. A specification begins with the keyword `message`, which is followed by an identifier indicating the name of the protocol considered. This identifier is then followed by one or more Zebu rule definitions (*message-body*).

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. The `statusLine` rule represents the same for a response. Header rules that appear in such a block represent headers that can appear only in either a request or response.

Header Definitions

A header declaration begins with the keyword `header`, which is followed by an `ident` indicating the name of the header considered. Lines whose beginning match the rule name are thus parsed as this particular kind of header. Alternative names can be provided between braces.

Regular Rule Definitions

Regular rule definitions are used to represent classic ABNF rules. A rule definition consists of a rule name followed by =. An alternation of successive elements then defines the rule body.

Layout Member

Each grammar element can be annotated by a `:` followed by an identifier, making the element available in the data structure representing a message. Optionally, a type attribute can be added both to cast the parsed element to a type other than string and to set constraints on it value.

```
spec          ::= message ident { message-body+ }
message-body  ::= request-block | response-block
                | header | zabnf
                | {constraint}
request-block ::= request { request-body+ }
request-body  ::= request-line | header | zabnf
                | {constraint}
request-line  ::= requestLine = elements ({attribute})*
response-block ::= response { response-body+ }
response-body ::= status-line | header | zabnf
                | {constraint}
status-line   ::= statusLine = elements ({attributes})*
header        ::= header ident ({ alt-ids })? = elements ({attributes})*
zabnf         ::= layout-attribute? ident = elements
                | ident =/ elements
elements      ::= 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
                | ( elements ) | [ elements ]
layout-member ::= : ident as type-attr?
type-attr     ::= uint8 | uint16 | uint32
alt-idents    ::= "ident" (/ alt-idents)?
```

Figure B.1: Zebu BNF Syntax

Attributes

Attributes can be added to grammar rules to indicate constraints or optimisation opportunities. Currently, four attributes are supported. `ReadOnly`, `mandatory` and `multiple` can only be applied to header rules. `ReadOnly` prevents modifications of an header that is supposed to be constant throughout the processing. `mandatory` indicates that the annotated header must be present in every message. Finally, `multiple` specifies that the header can appear several times in one message. The `lazy` attribute enables incremental parsing for any rule. The parsing described by these rules is only done when requested by the application.

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

Figure B.2: Attributes Specification

B.2 Constraints

A Zebu programmer can add various structural constraints, based on the natural language text, alongside the original ABNF protocol specification. Two kinds of constraints exist: value constraints and presence constraints. A *value constraint* express value dependencies; the value of an element is constrainted by the value of an other element. A *presence constraint* expresss presence depedencies; a header can become mandatory according to the presence or the value of some other message elements.

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

Figure B.3: Constraints Specification

Appendix C

The z2z Language

The z2z language provides three kinds of modules: protocol specification modules for defining the characteristics of protocols, message specification modules for describing the structure of protocol messages, and a message translation module for defining how to translate messages between protocols. We now present the concrete syntax of these modules in more detail using a BNF-like notation. `typewriter font` is used for keywords. *italic font* is used for non-terminals. Parentheses are used for grouping. A superscript $*$ denotes 0 or more occurrences of the preceding item. A superscript $+$ denotes 1 or more occurrences of the preceding item. A superscript ? indicates that the preceding item is optional. Identifiers (*ident*) and various other base type terms (*char-val*, *num-val*, etc.) are left unspecified.

C.1 Protocol specification module

The protocol specification (PS) module defines the properties of a protocol that a gateway should use when sending or receiving requests or responses. A PS module begins with the `protocol` keyword, which is followed by an identifier indicating the name of the protocol considered. This identifier is then followed by one or more declarations and block rule definitions. The grammar for the PS module is as follows, where a statement $statement\_base_{MT}$ refers to a statement from the MT module, as defined in Section C.3.2.

protocol ::= *extern_fun*$_{MT}^{*}$ `protocol` *ident* `{` *decl** *block** `}`
decl ::= *fftype ident* `;`
fftype ::= `int` | `fragment` | *fftype* `list`
block ::= `start {` *statement_base*$_{MT}^{+}$ `}`
| `request` *ident* `{` (*msgtypedef* `;`)* `}`
| `response {` (*response_msg* `;`)$^{+}$ `}`
| `flow = {` *ident* (`,` *ident*)* `}`
| `session_flow = {` *ident* (`,` *ident*)* `}`
| `sending request` *ident* `{` *statement_base*$_{MT}^{+}$ `}`
| `sending response` *ident* `(` `request` *ident* `)` `{` *statement_base*$_{MT}^{+}$ `}`
| `attributes {` (*attribute* `;`)* `}`
| `tcp {` *tcp_prototype* `}`
| `automata` *side* `{` *automaton*$^{+}$ `}`
tcp_prototype ::= *rtype* `tcp_connect ( ) ;`

C.1.1 Attribute block

The `attributes` block indicates the type of interaction with the network, in terms of the transport protocol used, whether requests are sent by unicast or by multicast, and whether responses are received synchronously or asynchronously.

```
attribute ::= transport = proto / integer
            | mode = mode
            | mode = mode / (multicast | unicast)
            | group = ipv4address
proto     ::= udp | tcp | tcp alive
mode      ::= sync | async
```

C.1.2 Requests and responses

The `request` block declares how to map messages to handlers. Similarly, the `response` block defines classes of responses. The expression $expression_{MT}$ refers to an expression from the MT module, as defined in Section C.3.3.

```
msgtypedef    ::= rtype ident when expression_MT
response_msg  ::= ident
                | ident = ident (| ident)*
rtype         ::= void | response
```

C.1.3 Automata

The `automata` block defines the state machines of a client (sending request) and a server (receiving requests) according to the protocol.

```
automaton  ::= automaton ident { (transition ;)+ }
side       ::= client | server
transition ::= lstate : ? ident (| ? ident)* -> state
             | lstate : ! ident (| ! ident)* -> state
             | lstate : epsilon -> state
             | lstate
lstate     ::= ident
             | start ident
             | final ident
state      ::= ident | ident . ident
```

C.2 Message specification module

The message specification (MS) module contains a description of the messages that can be received and created by a gateway. A message specification module provides a *message view* describing the relevant elements of incoming messages and *templates* for creating new messages.

```
msg ::= message ident { message_view template+ }
```

C.2.1 Message view

A message view describes the information derived from received messages that is useful to the gateway.

```
message_view  ::= read { message_place* }
message_place ::= qualifier visibility ty ident ;
qualifier     ::= mandatory | optional
visibility    ::= public | private
ty            ::= int | int8 | int16 | fragment | ty list
```

C.2.2 Templates

A template describes the information needed to create new messages. A template has three parts: the structural declarations, the template view, and the template text.

```
template      ::= request binary? template reqty ident implements template_def
                | binary? template ident implements template_def
                | response binary? template ident implements template_def
                | message binary? template ident implements template_def
implements    ::= (implements ident)?
reqty         ::= void | response
template_def  ::= { attribute* template_place* }
attribute     ::= magic = string ;
                | newline = \n | \r | \r\n ;
template_place ::= visibility ty ident ;
                | visibility ty ident = (string | integer) ;
```

C.3 Message translation module

The message translation (MT) module expresses the message translation logic of the gateway. This module consists of a set of handlers, one for each kind of relevant incoming request, as indicated by the protocol specification module.

```
program   ::= top_decl* mthd+
top_decl  ::= extern_fun
            | decl
mthd      ::= rtype ident ( ident_SOURCE request ident ) { decl* statement+ }
            | (void | response) ident_SOURCE _tcp_connect ( ) { decl* statement+ }
            | void ident_SOURCE _tcp_close ( ) { decl* statement+ }
rtype     ::= void
            | ident_SOURCE response
extern_fun ::= extern (int | void)? ident ( (funty (, funty)*)? ) ;
funty     ::= int | char * | ...
```

C.3.1 Declarations

A variable may be local to a handler, or may be declared outside of any handler if it it associated with a session. In the remainder of this section, $ident_{SOURCE}$ refers to the name of the source protocol, $ident_{TARGET}$ to the name of one of the target protocol, $ident_{PROTOCOL}$ to any other protocol name, and $ident_{function}$ to the name of a function.

```
decl      ::= ty decl_ident (, decl_ident)* ;
decl_ident ::= ident (= expression)?
ty        ::= ftype | bool
            | ident_SOURCE response | ident_TARGET response
            | ident_TARGET request
            | ident_PROTOCOL message
ftype     ::= int | fragment | eftype list
eftype    ::= int | fragment | ident_PROTOCOL message
            | eftype list
```

C.3.2 Statements

The grammar of statements is as follows:

```
statement ::= statement_base
            | ident = send ( nident ) ;
            | send ( nident ) ;
            | return ;
            | return nident ;
            | preturn nident ;
            | session_start ( ) ;
            | session_end ( ) ;
            | rearm ;
            | ident = ident_TARGET _tcp_connect ( ) ;
            | ident_TARGET _tcp_connect ( ) ;
            | ident_TARGET _tcp_close ( ) ;
            | foreach ( ftype ident = ident ) statement
            | break ;
            | ident_TARGET _set_host ( expression , expression ) ;

statement_base ::= ident = nexpression ;
                 | ident . ident = nexpression ;
                 | if ( expression ) statement (else statement)?
                 | { statement+ }
                 | ident_function ( (expression (, expression)*)? ) ;
nident         ::= ident
                 | ident (. ident)? ( keyword_args? )
keywords_args  ::= ident = naexpression (, ident = naexpression)*
```

C.3.3 Expressions

The grammar of expressions is as follows:

```
nexpression  ::= expression
               | ident . ident ( keyword_args )
               | ident ( keyword_args )
naexpression ::= expression
               | ident . ident ( keyword_args? )

expression ::= string
             | true | false
             | integer
             | ident | ident . ident
             | expression binop expression
             | ( expression )
             | ! expression
             | cons ( expression , ident )
             | head ( ident )
             | tail ( ident )
             | null ( ident )
             | empty ( ident . ident )
             | ident_function ( (expression (, expression)*)? ) ;
             | ident_TARGET _tcp_get_connection ( )
             | ident_TARGET _tcp_use_connection ( expression )
binop      ::= == | != | >= | <= | > | > | + | - | && | ||
```