Programmation orientée domaine pour les services télécoms: Concepts, DSL et outillage

Areski Flissi
LIFL/CNRS (UMR 8022)
59655 Villeneuve d'Ascq
areski.flissi@lifl.fr

Gilles Vanwormhoudt
Institut TELECOM
59655 Villeneuve d'Ascq
vanwormhoudt@telecom-lille1.eu

Abstract

Nous présentons dans cet article un langage dédié à la programmation des services télécoms et son outillage IDM au sein d'Eclipse. La particularité de notre langage est qu'il offre un certain nombre de constructions spécifiques au domaine des télécoms et qu'il permet aussi l'utilisation de constructions classiques d'un langage généraliste. Cette caractéristique permet de programmer à un niveau d'abstraction élevé et adapté au domaine, sans être limité dans l'expression de comportements complexes.

1 Introduction

Ces dernières années, la notion de langage dédié ou DSL (*domain-specific language*) [5] a connu un intérêt croissant dans le domaine du développement logiciel. Les DSLs sont des langages de spécification ou de programmation qui fournissent une notation et des constructions appropriées pour prendre en compte la spécification des besoins au plus près du niveau d'abstraction du domaine traité. Les avantages des DSLs sont d'accroitre la lisibilité, la concision et la sûreté des programmes. Ces avantages et l'existence de méta-outils facilitant de plus en plus leur construction ont conduit à l'émergence de DSLs dans de nombreux domaines [4].

Dans ce travail, nous nous intéressons à la définition de DSLs pour le développement d'applications et de services télécoms avancés basés sur le protocole SIP comme la visio-conférence ou la gestion d'appels téléphoniques avec gestion de présence et géolocalisation. En général, ces services sont complexes à concevoir car ils font intervenir plusieurs entités distribuées communiquant de manière asynchrone, concurrente et symétrique. Pour développer ces services, deux types d'approches existent. La première est d'utiliser des bibliothèques dans un langage de programmation généraliste comme par exemple [1, 6]. La seconde approche repose sur l'utilisation de langages dédiés comme par exemple les langages *CPL* [3], ECharts [8] et StratoSIP [7]. Ces deux approches offrent des avantages différents. La première approche est généralement plus souple et permet le développement de tout type de services mais nécessite une plus grand expertise en programmation, protocoles et systèmes distribués. De leur côté, les DSLs fournissent des abstractions et des opérations spécialisées de haut-niveau qui simplifient la conception des applications. Cependant, ils manquent d'expressivité pour implanter des services SIP sophistiqués.

Bien que nous ayons expérimenté la première approche, le travail présenté ici propose un DSL basé sur des concepts d'acteurs, de sessions et de rôles pour répondre spécifiquement aux différents problèmes posés par la conception des services télécoms, comme la gestion de plusieurs sessions multi-parties en parallèle et de longue durée. Comparé aux autres DSLs du domaine, la particularité de notre DSL est qu'en plus de ces concepts de haut-niveau, celui-ci offre également des constructions d'un langage de programmation généraliste, permettant ainsi d'exprimer tout type de comportement pour les entités participant au service. Dans la suite, nous donnons un aperçu des concepts du modèle de programmation sous-jacent. Puis, après avoir discuté des formes d'intégration de ces concepts dans un langage généraliste, nous décrivons le DSL. La méthode particulière utilisée pour construire ce DSL en réutilisant les constructions d'un langage généraliste ainsi que les outils fournis avec le DSL sont enfin présentés.

2 Du modèle de programmation au langage dédié

Concepts pour la programmation de services télécoms
Notre modèle de programmation, décrit plus en détail dans [9], repose sur les concepts d'acteurs, sessions et rôles. Un acteur représente une entité distribuée qui communique avec d'autres entités. Un acteur prend part à une ou plusieurs sessions et joue différents rôles au sein de ces sessions. Il doit contenir la logique d'activation des sessions et des rôles à la réception d'un message, c'est-à-dire les règles spécifiant quelle session et quel role sont concernés par la réception du message. La notion de *session* représente un échange persistent entre plusieurs acteurs. Elle est donc transverse aux acteurs. Dans notre modèle, une session est vue comme un ensemble de rôles qui interagissent. Enfin, la notion de rôle représente le comportement particulier d'un acteur au sein d'une session. Le rôle est la brique de base du modèle : il capture une logique métier récurrente. Par exemple, le comportement "`Envoyer réponse OK si réception requête INVITE`" peut être vu comme le rôle joué par un acteur répondant positivement à une demande d'établissement de session d'un autre acteur. La figure 1 illustre les principaux concepts de notre modèle de programmation avec un exemple de trois acteurs jouant différents rôles selon les sessions.

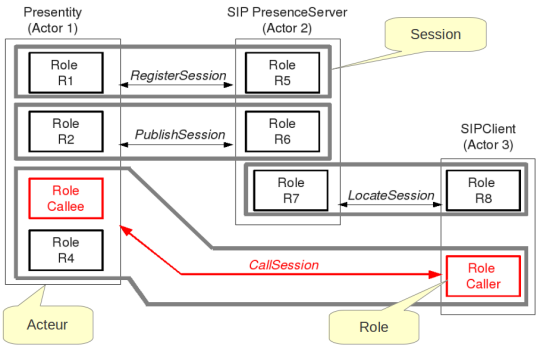

Figure 1: Les concepts d'acteurs, sessions et rôles de notre modèle

Intégration des concepts du domaine dans un langage généraliste
À partir du modèle de programmation décrit précédemment, nous avons cherché dans un premier temps à déterminer la façon la plus appropriée pour construire des programmes basés sur ces concepts en utilisant des langages objets généralistes comme Java. Ceci nous a conduit à étudier les trois approches suivantes :

i) Framework de classes
Une première façon d'intégrer les concepts du domaine dans les programmes est de construire un framework de classes correspondant aux différents concepts, ces classes étant prévues pour être spécialisées dans le cadre d'une application particulière. Une première expérimentation a aboutit à la création d'un framework avec des classes abstraites comme *ActorType*, *SessionType*, *RoleType*, *SessionPart*, *etc.*, dotées de comportement pour mettre en oeuvre notre modèle de programmation à l'exécution. L'intérêt de cette approche est qu'elle ne nécessite pas d'extension particulière du langage généraliste et qu'elle offre une souplesse assez importante pour écrire le code spécifique lié à l'application. Ses inconvénients sont que la spécialisation est souvent complexe à faire et que les aspects du domaine sont mélangés avec des aspects plus techniques, rendant la maintenance plus difficile.

ii) Annotations
Une seconde approche possible pour embarquer de façon plus explicite les concepts dans les programmes est d'utiliser le concept d'annotation offert par certains langages comme Java ou C♯. Une annotation permet d'ajouter des informations à certaines constructions de façon à spécialiser leur sens pour un domaine particulier. Dans nos travaux, nous avons défini un jeu d'annotations Java (*@actor, @session @sessionPart, @serverRole, etc.*) qui relie les concepts de notre modèle de programmation à certaines constructions Java. Nous avons aussi construit un outil interprétant ces annotations pour produire automatiquement le code spécialisant le framework précédent. Par rapport à l'approche précédente, les annotations permettent de donner un statut plus explicite aux concepts du domaine et d'obtenir un guide plus strict pour structurer les programmes selon le modèle de programmation. Concernant les inconvénients, cette approche contraint les possibilités pour exprimer et structurer les concepts du domaine et peut conduire à des correspondances inadaptées aux constructions du langage.

iii) Extension du langage
Une troisième et dernière approche possible pour embarquer de façon étroite les concepts du domaine dans un langage généraliste est de procéder à son extension. Avec cette approche (dite de *DSL interne*), les possibilités offertes dépendent des capacités d'extension du langage, qui peuvent aller d'une syntaxe étendue jusqu'à l'adaptation du noyau d'exécution. Récemment, plusieurs langages comme Ruby, Groovy, Closure ou Scala ont montré leur capacité à embarquer de façon plus ou moins fluide les concepts d'un domaine donné en DSL interne. Les avantages de cette approche sont de ne pas nécessiter la définition d'un nouveau langage complet et de pouvoir profiter du langage généraliste hôte et de ses outils de développement pour exprimer et tester des parties complexes. En revanche, composer les concepts du domaine avec les constructions du langage de façon cohérente et orthogonale peut être une tâche difficile. Un autre inconvénient réside dans la structuration des programmes qui reste celle du langage hôte. Dans nos travaux, nous avons laissé de côté cette approche pour nous concentrer sur la construction d'un DSL.

Présentation du langage dédié
Suite aux expérimentations précédentes, nous avons cherché à accroître encore plus la programmation avec les concepts du domaine, ce qui nous a conduit à concevoir un DSL spécifiquement adapté à la programmation des services télécoms. Ce DSL de programmation a été conçu en partant de notre modèle de programmation. Il permet de définir un service en décrivant les acteurs, sessions et rôles impliqués. La particularité de notre langage est qu'en plus d'offrir des constructions spécifiques au domaine des services télécoms pour structurer les programmes, il possède également des constructions empruntées au langage de programmation généraliste classique pour décrire les comportements.

Le code 1 illustre notre langage sur un exemple concret : un service de messagerie instantanée basée sur SIP. Dans cet exemple, deux acteurs interagissent selon le scénario suivant : un acteur *UAC* établit une session de communication avec un acteur *UAS*, puis les deux acteurs échangent des messages de type texte. Cet exemple fait intervenir un type de session que nous nommons *Chat*. Trois rôles sont identifiés au sein de cette session : un role *Caller* chargé d'envoyer une requête `INVITE` et de traiter une réponse `OK`, un rôle *Callee* chargé de répondre positivement à la réception d'une requête `INVITE`, et enfin, un rôle *Messagee* émettant des requêtes SIP de type `MESSAGE`.

Pour chacun des acteurs, il faut spécifier les rôles joués, ainsi que la logique d'activation des

```
service exemple {
    // session type definition
    session Chat {
        // role types definitio
        role Callee {
            receive INVITE { ...
            send OK { ... }
        }
        role Caller {
            send INVITE { ... }
            receive OK { ... }
        }
        role Messagee {
            send MESSAGE { ... }
            receive OK { ... }
        }
    }
```

```
// actor types definition
actor UAC {
    sessionPart CallerSession : Chat {
        play [roleinstance C1:Caller]
    }
}
actor UAS {
    sessionPart CalleeSession : Chat {
        play [C2:Callee, M1:Messagee]
        roleControl {
            when (INVITE) activate C2
            when (MESSAGE & isConnected ) activate M1
        }
        sessionControl {
            when (INVITE & isNotBusy ) activate
              CalleeSession
        }
    }
} // end service definition
```

Table 1: Le code DSL d'une application de messagerie instantanée SIP

sessions et des rôles (partie droite du code 1). Le mot clef `sessionPart` précise les sessions auxquelles participe un acteur, tandis que `play` permet d'indiquer quels rôles seront joués par l'acteur au sein de cette session. Plusieurs types de sessions, ainsi que plusieurs instances du même type de session peuvent cohabiter (la participation d'un acteur à une session étant identifiée via le mot clef `sessionPart` suivi d'un nom). La logique d'activation est un mécanisme qui permet de déterminer dynamiquement quels sont les sessions et rôles concernés par un événement[1]. Dans notre exemple, un seul type de session existe, le mot clef `sessionControl` permet d'indiquer que lorsque l'acteur *UAS* reçoit un message SIP de type `INVITE`, cela concerne la participation de cet acteur à une session de type *Chat*.

L'avantage de proposer un tel DSL de programmation est double. D'une part, le concepteur tire les bénéfices liées à l'utilisation d'un DSL : syntaxe permettant de guider le concepteur spécialiste du domaine, concepts de haut niveau facilitant la concision des programmes et leur maintenance, contrôles de typage spécifiques au domaine. D'autre part, la possibilité offerte d'utiliser au sein du DSL des constructions d'un langage généraliste fait bénéficier le concepteur des avantages d'un GPL : réutilisation de constructions standards (types, boucles, etc.) pour exprimer des aspects algorithmiques variés, capitalisation de l'expérience des développeurs pour des parties complexes et récurrentes, etc.

3 Outillage

Construction du langage dédié

Pour construire notre DSL, nous avons procédé selon une méthode particulière qui consiste à réutiliser les concepts d'un langage généraliste existant quand cela est pertinent et possible pour définir ceux du DSL. Il convient de souligner que dans le cas présent, le mode de réutilisation utilisé ne vise pas une extension du langage existant mais une mise en oeuvre du DSL avec un minimum d'effort, notamment pour les parties dédiées à la description des comportements (boucles, conditionnels). Avec cette méthode, nous avons procédé de plusieurs manières pour définir les concept spécifiques de notre DSL. Certains concepts ont été définis sans faire référence

[1] Dans une application SIP classique, ceci implique notamment une gestion d'états ainsi qu'une analyse du contenu du message par le programmeur.

à ceux du langage existant. C'est le cas par exemple des constructions *session* et *play*. D'autres concepts ont été obtenus par héritage des concepts existants, comme *actor*, *role* et *sessionPart* qui héritent du concept de classe pour inclure des définitions d'attributs et d'opérations. Enfin, un dernier ensemble de concepts du DSL a été défini en agrégeant des concepts existants. C'est le cas par exemple des constructions *send*, *receive* pour leur corps mais aussi de la construction *when* pour une partie de la condition.

Cette méthode de construction du langage a été rendue possible grâce à l'utilisation des plugins EMFText et JaMoPP [2] dans l'environnement EMF d'Eclipse. EMFText permet à partir d'un méta-modèle existant et d'une spécification de syntaxe concrète d'un langage de générer du code Java implantant un transformateur de texte valide en un modèle conforme, ainsi qu'un éditeur textuel adapté. JaMoPP est une bibliothèque construite avec EMFText. Cette bibliothèque implante un méta-modèle exhaustif de Java ainsi que sa syntaxe concrète, un éditeur et des fonctionnalités de validité sémantique, notamment celles liées au typage. EMFText présente comme caractéristique intéressante de permettre de définir les éléments d'un langage à partir de ceux d'un autre.

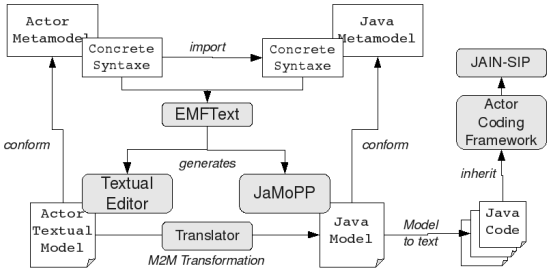

Figure 2: Architecture des plugins Eclipse de notre outillage

Comme on peut le voir sur la figure 2, nous avons définit le méta-modèle de notre DSL en utilisant une partie des concepts du méta-modèle Java définis par JaMoPP. La grammaire de notre langage inclut des règles syntaxiques du langage Java. Il devient, par ce biais, possible d'inclure des parties de code Java dans notre DSL mais aussi d'utiliser les fonctionnalités de validité sémantique du langage Java fournies par JaMoPP pour vérifier ces parties.

Atelier IDM dans l'environnement Eclipse

Autour de ce langage, nous avons élaboré un outillage dans l'environnement Eclipse, afin de faciliter, selon une approche guidée par les modèles, la conception de services télécoms. Il se compose d'un ensemble de plugins facilitant le développement d'applications SIP sous Eclipse :
– Un éditeur de modèles textuels facilitant l'écriture de programmes basés sur notre DSL. Celui-ci a été généré grâce au plugin EMFText et incorpore des fonctionnalités d'aide à la complétion et d'analyse syntaxique et sémantique.
– Un éditeur graphique pour concevoir des sessions et des acteurs avec une vue de haut-niveau, réalisé grâce aux plugins EMF/GMF.
– Un traducteur vers du code Java spécialisant un framework de programmation s'exécutant au dessus de l'architecture JAIN-SIP[1]. Ce traducteur a été conçu en s'appuyant sur JaMoPP. Afin d'obtenir le code exécutable à partir d'un modèle conforme à notre méta-modèle, nous appliquons une transformation M2M vers un modèle JaMoPP, puis une transformation de ce modèle en code Java.
– Une bibliothèque de roles/sessions réutilisables.

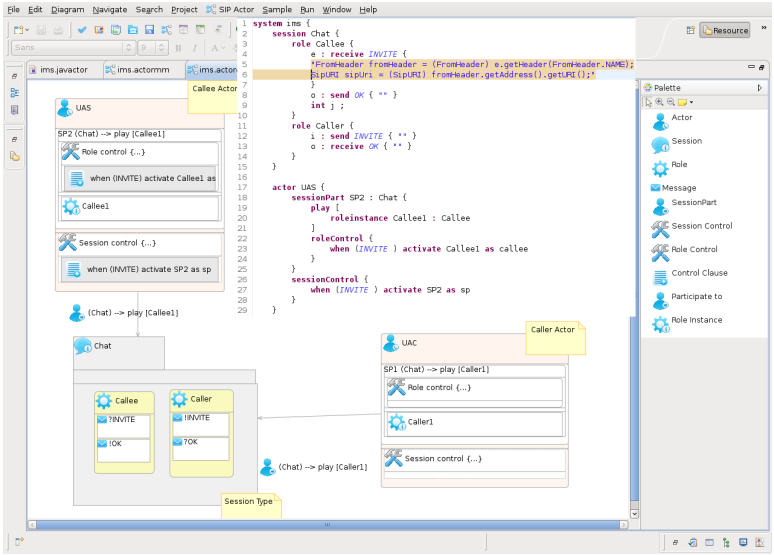

Figure 3: Vue Eclipse de notre environnement de développement

La figure 3 représente une vue d'Eclipse de la conception de l'exemple précédent. À partir du modèle graphique, l'environnement offre la possibilité de générer le modèle textuel, c'est-à-dire le code DSL. Celui-ci peut alors être complété directement afin d'y intégrer notamment le code Java implémentant l'action à déclencher lors de la réception ou l'envoi d'un message. La synchronisation entre les différentes vues d'un même modèle (graphique, textuel ou arborescent) permet de tirer profit des avantages de chacune d'entre elles. Enfin, à partir du code DSL, nous obtenons, après transformation, le code Java exécutant le service au dessus de notre framework.

References

[1] The JAIN-SIP Specification. http://jcp.org/aboutJava/communityprocess/final/jsr032/index.html.

[2] TU Dresden: Software Technology Group. EMFText and JaMoPP. http://emftext.org, http://www.jamopp.org.

[3] J. Lennox, X. Wu, and H. Schulzrinne. RFC3880 : Call Processing Language (CPL) : A Language for User Control of Internet Telephony Services. Technical report, 2004.

[4] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[5] M.Fowler. *Domain-Specific Languages*. Addison-Wesley, 2011.

[6] A. Moizard. The GNU oSIP library.

[7] P.Zave, E.Cheung, G. Bond, and T. Smith. Abstractions for Programming SIP Back-to-Back User Agents. In *Proceedings of IPTComm'09*, 2009.

[8] T.Smith and G.Gregory W. Bond. ECharts for SIP Servlets: a state-machine programming environment for VoIP applications. In *Proceedings of IPTComm'07*, 2007.

[9] Gilles Vanwormhoudt and Areski Flissi. Session-based Role Programming for the Design of Advanced Telephony Applications. In *Proceedings of DAIS'11*, 2011.