UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

MÉTHODE FORMELLE DE DÉTECTION D'INTERACTIONS POUR LES POLITIQUES DU CONTRÔLE D'APPEL TÉLÉPHONIQUE

MÉMOIRE
PRÉSENTÉ
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE

PAR

LAYOUNI AHMED FADHEL

NOVEMBRE 2007

À ma famille et spécialement mon frère Mohamed !
Sans vous rien de cela ne serait possible.

Et à tous mes amis !
Pour avoir été de bons et vrais amis, merci pour tout.

Remerciements

C'est avec un grand plaisir que je réserve cette page en signe de gratitude et de profonde reconnaissance à tous ceux qui m'ont aidé à la réalisation de ce travail.

Je tiens à exprimer mes sentiments de gratitude et de reconnaissance au professeur Luigi LOGRIPPO pour son encadrement, son soutien et ses conseils judicieux tout au long du projet et pour l'ambiance amicale que j'ai trouvée au sein de notre équipe de recherche.

Ce travail a été subventionné en partie par le Conseil de Recherches en Sciences Naturelles et en Génie du Canada.

Mes remerciements vont à tous les enseignants de l'UQO pour la qualité de la formation qu'ils m'ont fournit tout au long de mon cursus de maîtrisard en informatique.

Je ne peux finir sans remercier Professeur Michal IGLEWSKI et Professeur Andrzej PELC, les membres du jury, d'avoir bien voulu s'intéresser à mon travail et de prendre part dans son enrichissement et évaluation.

J'en suis sincèrement honoré.

Table des matières

Liste des figures

Liste des tableaux

Liste des abréviations, sigles et acronymes

ACCENT Advanced Component Control Enhancing Network Technologies

APPEL ACCENT Project Policy Environment/Language

CFA Call Forwarding Always

CFB Call Forwarding when Busy

CPL Call Processing Language

CW Call waiting

DNS Domain Name Service

IP Internet Protocol

LESS Language for End System Services

OCS Originating Call Screening

OWL Web Ontology Language

PSTN Public Switched Telephone Network

SIP Session Initiation Protocol

SIPC Columbia SIP User Agent

SOAP Simple Object Access Protocol

UA User Agent

UAC User Agent Client

UAS User Agent Server

UDDI Universal Description Discovery and Integration

VoD Video on Demand

VoIP Voice over IP

WSDL Web Services Definition Language

Résumé

La conception et l'implémentation de nouvelles fonctionnalités occupent une place primordiale dans le domaine de la télécommunication. La création rapide de nouveaux services et le recours aux fonctions personnalisées, visant à mieux répondre aux besoins des clients, rendent cette tâche fondamentale. Cependant, la gestion de ces fonctionnalités est une tâche complexe, vu le nombre croissant des services offerts aux consommateurs, leurs complexités et le fait que plusieurs fonctionnalités peuvent interagir sur un même terminal, ainsi qu'entre plusieurs terminaux. La téléphonie sur Internet permet d'exécuter des services sur des terminaux intelligents offrant plus de fonctionnalités aux usagers. Ces services peuvent générer des interactions de fonctionnalités possiblement indésirables. Les systèmes LESS (Language for End System Services) et APPEL (ACCENT Project Policy Environment/Language) sont des outils de gestion des services téléphoniques, largement utilisés en pratique.

Dans ce travail, nous proposons une méthode formelle pour la détection des comportements conflictuels des systèmes LESS et APPEL. Notre méthode se base sur l'analyse des pré et post-conditions des actions. Nous utilisons l'outil Alloy Analyzer pour modéliser les systèmes et les politiques ainsi que pour détecter des interactions.

Abstract

The design and implementation of new features is an important trend in telecommunication. The fast creation of new services and the increasing use of personalized functions, aiming to better satisfy users requirements, make this task fundamental. Given the growing number of services offered to users, their complexity, and the fact that several features may interact on the same terminal, as well as on different terminals, the task of managing such features has become extremely complex.

Telephony over the Internet (a.k.a., Voice over IP) makes it possible to carry out services on intelligent terminals, offering even more features to users. With these extra features, comes also the possibility of unwanted feature interactions.

LESS (Language for End System Services) and APPEL (ACCENT Project Policy Environment/Language) are widely used management tools for telephony services.

In this work we propose a formal method to detect conflicting behaviours in the LESS and APPEL systems. Our method is based on the analysis of actions' pre and post-conditions. We use the Alloy Analyzer tool to model systems, policies, and to detect their interactions.

Chapitre 1

Introduction et motivation

L'utilisation d'Internet en tant que plateforme pour la livraison de données et l'augmentation de sa bande passante, rendent possible la création d'un grand nombre de services téléphoniques qui s'exécutent sur les protocoles Internet. Malheureusement, l'expérience nous fait connaître que ces services peuvent interagir de manière non-désirable. Ce problème était déjà connu dans la téléphonie conventionnelle, Public Switched Telephone Network (PSTN), basée sur les commutateurs de circuits, qui cependant n'avait pas la possibilité de gérer un grand nombre de fonctionnalités ainsi que leurs interactions.

La téléphonie sur Internet offre des communications multimédia, des services intégrés et permet le développement rapide de services personnalisés. De ce fait, la téléphonie sur Internet offre plus de fonctionnalités et génère de nouvelles interactions de fonctionnalités. La téléphonie sur Internet peut empêcher certaines interactions de fonctionnalités puisqu'elle offre une plus grande facilité dans la gestion des problèmes liés aux réseaux et aux systèmes de décisions. Cependant la gestion des interactions de fonctionnalités devient une tâche de plus en plus compliquée et suscite l'émergence de plusieurs techniques pour empêcher et résoudre les interactions indésirables.

1.1 Fonctionnalités et interactions de fonctionnalités

Cette section introduit des définitions informelles de certains concepts de fond dans notre domaine de recherche. Des définitions plus formelles et spécifiques à notre étude seront présentées dans les chapitres 4 et 5.

"Les logiciels des systèmes de télécommunication sont composés d'un noyau qui fournit les préférences de base attendues et d'un ensemble d'entités auxiliaires, appelées fonctionnalités [1]."

Une fonctionnalité est une entité optionnelle ou complémentaire à un système téléphonique existant. Les fonctionnalités permettent le changement du comportement du système téléphonique de base, afin de mieux répondre aux besoins des utilisateurs.

"Les fonctionnalités offrent la possibilité aux usagers de gérer le contrôle de leurs appels téléphoniques." [2]

L'évolution de l'appel téléphonique est une succession d'événements qui font passer le système téléphonique d'un état à un autre. L'activation d'une fonctionnalité dépend de l'état actuel du système et de l'événement qui se produira. Des exemples de fonctionnalités bien connus sont le renvoi d'appel (Call Screening), la liste noire (Call Forwarding) et l'appel en attente (Call Waiting).

"Les interactions de fonctionnalités peuvent être définies informellement comme des violations des contraintes (ou des intentions) causées par la combinaison de plusieurs fonctionnalités" [21].

Considérons un client B abonné au service CFA (Call Forward Always). Ce service permet de renvoyer tous les appels rentrant à l'adresse (ou le numéro) d'un autre abonné (C par exemple).

Considérons maintenant un autre client A abonné au service OCS (Originating Call Screening). Ce service permet à son utilisateur de filtrer tous les numéros qu'il ne désire pas appeler. Si le client A rentre le numéro d'un client C dans sa liste de numéros filtrés, alors A ne devrait pas pouvoir appeler C.

Considérons maintenant la situation suivante :

Le service CFA de B est actif et tous ses appels sont renvoyés vers C. Supposons que A essaye de joindre B. L'appel de A sera acheminé vers B, puis renvoyé vers C. Par

conséquent, le client A, malgré le fait qu'il ait activé sa fonctionnalité OCS, communiquera quand même avec C. Ceci est illustré dans la figure 1.1. Il est clair dans notre exemple, que le conflit est dû, entres autres, au manque de précision dans la description des fonctionnalités.

Dans ce travail, nous considérons les interactions indésirables comme des conflits entre les fonctionnalités impliquées.

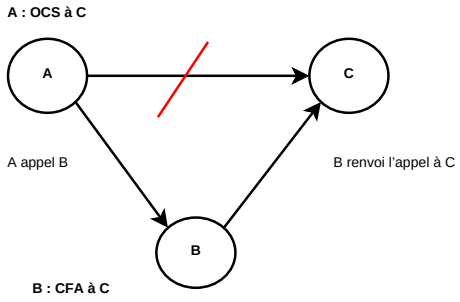

FIG. 1.1 – Interaction de fonctionnalités

Selon [14], les situations suivantes peuvent être considérées comme interactions de fonctionnalités :

- Non-associativité de composition des fonctionnalités.
- Incohérence logique.
- Comportement non-déterministe.

La figure 1.1 montre un exemple d'incohérence logique entre deux fonctionnalités.

Un exemple classique de non-associativité de composition des fonctionnalités est la composition des deux fonctionnalités (CW : Call Waiting et CFB : Call Forwarding when Busy) [25]. Quand un appel tente de joindre une ligne occupée, la fonctionnalité CW alerte l'appelé en générant une tonalité d'appel en attente. Tandis que la fonctionnalité CFB renvoie les appels rentrants d'un utilisateur occupé à un numéro alternatif. Si un même utilisateur est abonné à ces deux fonctionnalités, que se passe-t-il en cas d'appel rentrant quand l'utilisateur est occupé ? Si le système donne la priorité à la fonctionnalité CW, alors la fonctionnalité CFB sera discriminé et vice versa. On peut retrouver

plusieurs exemples d'interactions de fonctionnalités catégorisés par nature, par cause de l'interaction et par genre de fonctionnalité dans [6].

1.2 Politiques et interaction de politiques

"Les politiques saisissent les buts de haut-niveau afin de les mettre en vigueur automatiquement" [28] .

Dans la téléphonie sur Internet, les fonctionnalités correspondent à des politiques d'usagers. Par exemple, refuser tout appel rentrant provenant d'une certaine personne. Donc, une politique est une information qui modifie le comportement d'un système de base. Dans ce contexte, les fonctionnalités ne sont plus invoquées ou stimulées seulement par les utilisateurs ou l'état de l'appel. L'interaction de politiques est le changement du comportement d'une politique dû à la présence d'une autre politique.

Les politiques sont des règles (ou contraintes) conçues pour contrôler un système dynamiquement en exécutant certains ensembles d'actions. Les systèmes de politiques sont utilisés dans les domaines de contrôle d'accès, de qualité de services, en sécurité informatique et dans les systèmes de gestion. Dans toutes ces applications, les politiques sont créées et gérées par l'administrateur du système.

Dans cette étude, nous ne faisons pas la distinction entre les politiques d'usagers et les politiques d'administrateurs.

Dernièrement, plusieurs langages de politiques ont été développés dans le but de décentraliser le contrôle du comportement du système, de donner plus de droits aux utilisateurs et d'automatiser la gestion du système. Ces nouvelles flexibilités ont l'avantage de laisser l'usager faire son propre choix de services satisfaisant ses besoins.

Puisque les politiques sont définies d'une façon décentralisée, le potentiel des interactions entre politiques est plus élevé qu'avec les systèmes basés sur les fonctionnalités simples, comme dans la téléphonie traditionnelle. La flexibilité offerte par les systèmes de politique est contrebalancée par le grand nombre de conflits qui peuvent être générés.

Nous citons dans ce qui suit un exemple illustrant les interactions entre politiques.

Exemple : Interaction de politiques

Fonctionnalités impliquées

1. La messagerie vocale

   Le service de messagerie vocale de base, renvoi les appels à la messagerie, suite à l'une des fonctionnalités suivantes :

   - Call forward - busy line (ligne occupée).
   - Call forward - don't answer (en cas de non-réponse).
   - Call forward - variable ou paramètré. (Exemple : renvoyer les appels rentrants, les Samedis et Dimanches, sur le téléphone cellulaire).

   Ce service permet aux appels se terminant à une ligne occupée, de se réorienter à la messagerie vocale de l'appelé. Cette fonctionnalité exige qu'un terminal puisse identifier un service de messagerie, c'est-à-dire un serveur de messagerie vocale.

2. Renvoi de l'appel en cas de non réponse

   Cette fonctionnalité renvoie les appels rentrants, quand le terminal de l'appelé ne répond pas après un nombre de cycles de sonorité spécifié par l'utilisateur à une nouvelle adresse.

Politiques

- Transférer tout appel provenant de mon patron sur mon cellulaire sinon l'accepter si l'appelant n'est pas mon patron. (voir figure 1.2).
- Renvoyer les appels en provenance de Alice à ma messagerie vocale, sinon accepter tout autre appel (voir figure 1.3).

Interaction de politiques

L'activation de plusieurs politiques par un même usager peut conduire à des situations conflictuelles. La figure 1.4 met en relief un exemple d'interaction de fonctionnalités causé par un non déterminisme. Que se passe-t-il si Alice est mon patron ? Deux fonc-

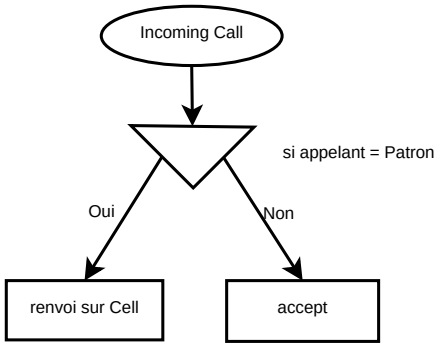

FIG. 1.2 – Politique de renvoi sur un autre terminal

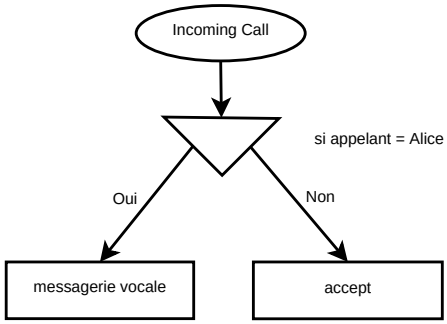

FIG. 1.3 – Politique de renvoi sur une messagerie vocale

tionnalités incompatibles sont invoquées. Dans ce cas précis, il faut identifier le probléme et aussi préciser quelle fonctionnalité est prioritaire par rapport à l'autre.

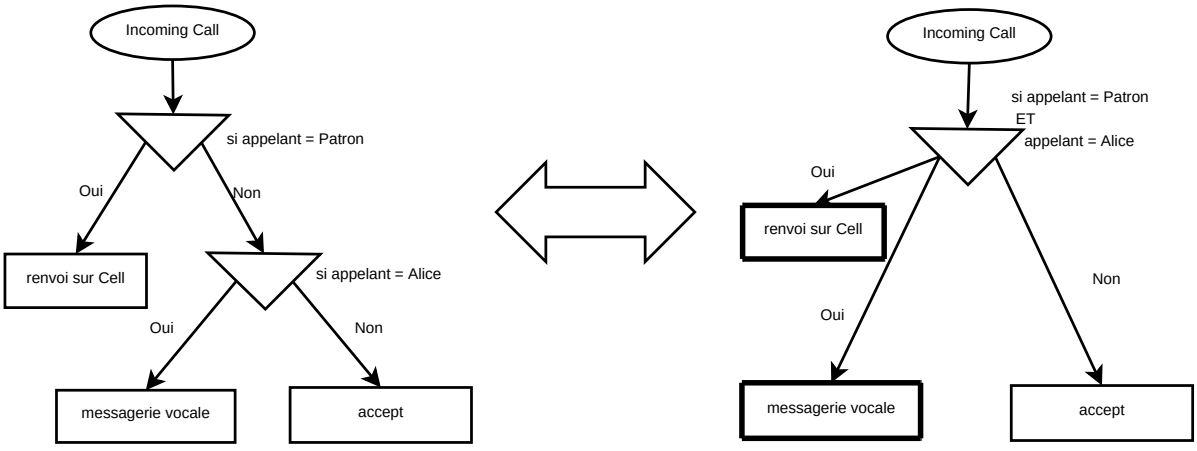

FIG. 1.4 – Conflit entre politiques

1.3 Contribution visée

Dans ce projet, nous étudions le problème de l'interaction de fonctionnalités, comme le résultat d'incohérences logiques entre les actions en concurrence, pendant l'exécution de différentes fonctionnalités téléphoniques.

Par exemple, par rapport à la figure 1.4, considérons les actions "messagerie vocale" (c.f. `redirect`) et "renvoi sur cell" (c.f. `transfer`). Ces deux actions, qui peuvent être invoquées en même temps, ont des post-conditions qui sont en contradiction, comme nous verrons en détail plus tard. Cette contradiction peut causer une interaction de fonctionnalités indésirable.

Nous présenterons ultérieurement la technique de détection d'interaction de fonctionnalités introduite dans [20]. Cette technique sera démontrée sur deux cas d'études, les systèmes Less et Appel, qui seront décrits respectivement aux chapitres 4 et 5.

Par rapport au système Less, nous envisageons fournir les contributions suivantes :

- Développement d'une méthode logique pour la représentation des règles téléphoniques.
- Formalisation des interactions possibles.
- Développement d'une méthode formelle de détection des interactions indésirables pour une seule règle. Le but de cette contribution est de permettre aux usagers non-expérimentés d'assurer la validité sémantique de leurs besoins.
- Développement d'une méthode de détection des interactions pour une paire de règles.
- Finalement nous menerons une évaluation expérimentale pour la détection des interactions du système.

Quand au système Appel, nous convoitons les objectifs suivant :

- Spécification des politiques de control d'appels téléphoniques.
- Définition des pré et post-conditions des actions.

- Identification des comportements conflictuels entre les politiques ; les causes et les types des interactions indésirables.
- Développement d'une méthode formelle de détection et d'analyse des cas conflictuels pour une paire de politiques.
- Diagnostic des résultats expérimentaux de notre méthode par rapport à la méthode de détection du concepteur du système APPEL.

Pour effectuer ces formalisations et analyses, nous avons utilisé le système Alloy Analyzer, qui n'a jamais été utilisé dans ce contexte.

La contribution de ce travail est essentiellement une méthode de détection des interactions basées sur les incohérences entre pré et post-conditions des actions.

1.4 Organisation de ce mémoire

Ce mémoire est organisé comme suit.

Chapitre 1 : Introduction et motivation .

Ce chapitre expose les nouvelles tendances et spécificités de la téléphonie sur Internet par rapport à la téléphonie traditionnelle. Nous introduisons d'abord les concepts de fonctionnalités et d'interaction de fonctionnalités. Nous soulignons ensuite les problèmes que peuvent causer les interactions de fonctionnalités. Nous expliquons dans un deuxième temps l'utilité d'exprimer les fonctionnalités sous forme de politiques pour faciliter la détection de conflits. Enfin, nous décrivons les contributions du mémoire.

Chapitre 2 : Cadre conceptuel .

Ce chapitre délimite les types de systèmes téléphoniques visés par cette étude. Nous introduisons également les services web, ainsi que les concepts de SIP (Session Initiation Protocol) et de CPL (Call Processing Language).

Chapitre 3 : État de l'art .

Ce chapitre présente les principales méthodes existantes pour détecter les interactions de fonctionnalités et de politiques téléphoniques.

Chapitre 4 : Premier cas d'étude : LESS et interactions dans LESS .

Dans ce chapitre, nous développons notre méthode formelle de détection des conflits. Nous donnons ensuite une description du système LESS et de ces éléments. Enfin, nous discutons les différents conflits entre actions qui ont été détectées pour le système LESS.

Chapitre 5 : Deuxième cas d'étude : APPEL et interaction dans APPEL .

Ce chapitre introduit le système APPEL. Nous élaborons d'abord les pré et post-conditions des actions du système APPEL. Ensuite, nous décrivons une méthode similaire à celle utilisée avec LESS pour analyser le système APPEL.

Chapitre 6 : Méthode logique pour la détection des interactions .

Ce chapitre présente d'abord l'outil de modélisation et de vérification Alloy Analyzer. Nous donnons ensuite une explication de la logique et la sémantique de l'outil Alloy nécessaires pour effectuer l'analyse des conflits dans LESS et APPEL via les méthodes présentées dans les chapitres 4 et 5 respectivement.

Chapitre 7 : Conclusion .

Ce chapitre résume la contribution de ce mémoire. Nous présentons ensuite quelques directions possibles pour des travaux futurs.

Chapitre 2

Cadre conceptuel

2.1 Introduction

Les interactions de fonctionnalités qu'on vise à détecter surviennent dans la couche application qui se trouve au dessus des couches CPL (Call Processing Language), SIP (Session Initiation Protocol) et TCP (Transmission Control Protocol) ou UDP (User Datagram Protocol) respectivement. Les "services web" sont un exemple typique de composants de la couche application (c.f., LESS, APPEL, etc.).

La couche application se distingue par une grande variété de services offerts aux usagers. Cette multitude de services et d'applications exécutées en concurrence peut générer des conflits suite à des incohérences logiques. C'est la raison pour laquelle nous nous intéressons à l'étude des conflits qui se produisent dans la couche application.

Dans ce travail, nous considérons deux exemples de services de la couche application : les systèmes LESS et APPEL utilisés dans le domaine de la téléphonie sur Internet.

La figure 2.1 présente les différentes couches impliquées pour qu'une communication téléphonique réussit. En effet, LESS utilise les services de la couche CPL. LESS et CPL utilisent les services de la couche SIP. La couche SIP se situe au dessus de la couche de transport. La couche SIP peut utiliser les protocoles TCP ou UDP.

Dans ce qui suit, nous présentons un aperçu des services web, du protocole SIP et du langage CPL.

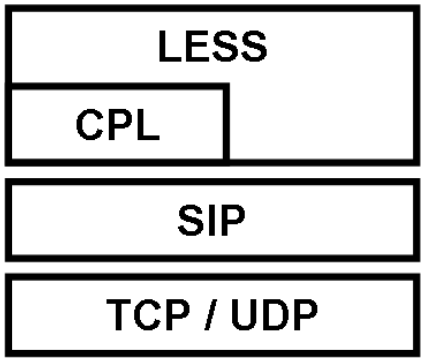

FIG. 2.1 – Couches de la téléphonie sur Internet

2.2 Les services web

L'architecture des services web est introduite par le World Wide Web Consortium [3]. Les services web sont des applications autonomes et modulaires qui peuvent être conçues, publiées, localisées et invoquées depuis un réseau, souvent, le web. La téléphonie sur Internet avec ses services représentent un exemple typique des services web.
La téléphonie sur Internet permet l'intégration de nouveaux services. Les services de la téléphonie traditionnelle, tels que le blocage d'appel, le renvoi d'appel et le transfert d'appel, peuvent être améliorés en combinant la téléphonie traditionnelle avec les services d'annuaires, les services de présence, le Web et la transmission de messages instantanée [23].

Par exemple, les utilisateurs peuvent avoir des appels réorientés aux pages web ; la transmission multimédia est utilisée pour enregistrer les messages vocaux. Les communications peuvent inclure en plus de la voix, la vidéo, le partage d'application, etc.

"Un Service Web est un programme informatique permettant la communication et l'échange de données entre applications et systèmes hétérogènes dans des environnements distribués. Il s'agit donc d'un ensemble de fonctionnalités exposées sur Internet ou sur un Intranet, par et pour des applications ou machines, sans intervention humaine, et en temps réel" [35].

Les services web peuvent être décrits en utilisant le langage Web Services Definition Language (WSDL) qui est un sous ensemble de XML. Cette description couvre

tous les détails nécessaires pour interagir avec les services, y compris les formats des messages, les protocoles de transport et l'emplacement. En outre cette description peut être publiée dans la base d'enregistrement, en utilisant le standard Universal Description Descovery and Integration (UDDI), où elle peut être trouvée ultérieurement par le service de requête. Les applications utilisent le Simple Object Access Protocol (SOAP) comme protocole de communication, dans le but de permettre leurs interactions avec les services web. Les messages SOAP sont supportés par le protocole HTTP. On peut utiliser toutefois d'autres protocoles de transport. L'interface du service web cache les détails d'implémentations du service, en lui permettant d'être utilisé indépendamment des plateformes matérielles ou logicielles et du langage de programmation utilisé. Ces références [32, 34, 30, 33] peuvent être consultées pour les spécifications XML, WSDL, UDDI et SOAP.

2.3 Le protocole Session Initiation Protocol (SIP)

Un des protocoles utilisés dans les services web est le protocole Session Initiation Protocol (SIP). Développé par Internet Engineering Task Force (IETF) [25]. SIP est un protocole de contrôle au niveau application, utilisé pour la coordination et la gestion des sessions, c'est-à-dire il permet aux utilisateurs d'établir, de modifier et de terminer les sessions des communications téléphoniques. Actuellement, il est utilisé dans la gestion des sessions multimédias, comme la voix sur Internet (VoIP), la conférence vidéo et la vidéo sur demande (VoD).

SIP est un protocole basé sur des scripts XML, architecturé sur les protocoles standards d'Internet, à savoir, Internet Protocol (IP), le Transmission Control Protocol (TCP) et le User Datagram Protocol (UDP).

Description du protocole SIP

Le modèle client/serveur est employé comme paradigme de communication. Une demande du client est répliquée par une réponse du serveur. SIP indique également le comportement de plusieurs entités du réseau : les User Agents, les Registrars, les Location Services et les SIP Proxys [25]. Le User Agent est un terminal capable d'établir et de terminer une session. Par conséquent, il peut agir en tant que client et serveur (paradigme de peer-to-peer). La communication s'établit entre un UAC (User Agent

Client) situé sur le terminal appelant et un UAS (User Agent Server) situé sur le terminal appelé. Chaque partie communicante est identifiée par une adresse de format URL (Uniform Resource Location) qui se présente sous la forme `sip ://user@domain`.

Les informations relatives aux User Agents pour un domaine spécifique sont stockées dans le Location Service [25]. Le User Agent inscrit son emplacement en utilisant un Registrar. Ce dernier, envoie ces informations au Location Service, qui agit comme une sorte de base de données contenant les adresses des User Agents. Si l'appelant connaît l'adresse de l'appelé, il peut le contacter directement. Si l'adresse de l'appelé n'est pas connue, l'adresse URL est utilisé conjointement avec un SIP Proxy servant d'intermédiaire.

Le SIP Proxy est une entité du réseau qui contrôle le flot de messages en les acheminant à l'entité (qui peut être un Proxy ou un User Agent) la plus proche qui est connue par le Proxy. Les Proxys utilisent le Service Location pour cette tâche : si l'appelé se trouve dans le même domaine que le Proxy, le Service Location est capable de retourner une adresse finale au User Agent. Si l'appelé n'est pas dans le même domaine, le message est transféré au Proxy responsable du domaine spécifique. Les adresses des Proxys responsables peuvent être obtenues par accès aux enregistrements du DNS (Domain Name System) [23].

Pour implanter la logique décrite précédemment, SIP utilise plusieurs messages de requêtes appelés aussi méthodes et six classes de messages de réponses. Dans ce qui suit, sont présentés brièvement les messages de requêtes et quelques messages de réponses. Plus de détails peuvent être trouvés en consultant ces références [25, 15, 16].

Les messages de requêtes

Ces messages de requêtes requièrent qu'une action particulière soit prise par un autre User Agent ou par un SIP proxy. Ces requêtes sont :

- **REGISTER** : cette requête est utilisée par un User Agent désirant enregistrer son adresse IP auprès du SIP Proxy auquel il est relié.
- **INVITE** : cette requête indique que l'utilisateur correspondant à l'URL spécifié est invité à participer à une session.

- **ACK** : cette requête confirme que le terminal appelant a reçu une réponse définitive à sa requête INVITE et que la session peut débuter.
- **BYE** : cette requête est utilisée pour mettre fin à une session préalablement établie.
- **CANCEL** : cette requête peut être utilisée par un User Agent ou par un SIP Proxy afin d'annuler une requête non validée par une réponse finale.
- **OPTIONS** : cette requête est utilisée pour s'informer sur la disponibilité ou la capacité d'un User Agent.

Les messages de réponses

Un message de réponse est généré par un UAS ou par un SIP Proxy en réponse à une requête transmise par un UAC. Nous introduisons dans ce qui suit les six classes de réponses.

1. **1xx Informational** : cette classe de réponses indique le statut d'un appel avant qu'il ne soit complet.
   - **100 Trying** : un SIP Proxy transmet cette réponse à l'appelant pour lui faire part qu'il tente de contacter l'appelé.
   - **180 Ringing** : la requête INVITE a été reçue et le UAS est entrain de transmettre une alerte de type sonnerie. Si le UAS répond automatiquement, cette réponse n'a pas besoin d'être transmise.

2. **2xx Succes** : cette classe de réponses confirme le succès d'une requête.
   - **200 OK** : cette réponse est transmise lorsqu'une requête de type INVITE est acceptée. Ce message est également transmis en réponse aux requêtes REGISTER, BYE et CANCEL.

3. **3xx Redirection** : message de redirection transmis par un SIP Proxy.
   - **300 Multiple choices** : cette réponse retourne au UAC un message contenant plusieurs localisations possibles pour le terminal appelé.
   - **305 Use Proxy** : cette réponse retourne l'adresse URL d'un autre SIP Proxy nécessaire pour joindre le terminal appelé.

4. **4xx Client Error** : classe retournant un message d'échec de la requête dû à une erreur du client.
   - **400 Bad request** : la requête n'a pas été comprise par le serveur.

5. **5xx Server Error** : classe retournant un message d'échec de la requête dû à une erreur du serveur.
   - **500 Server Internal Error.**
   - **503 Service Unavailable** : la requête demandée ne peut pas être traitée temporairement.

6. **6xx Global Error** : cette classe de réponses indique que le SIP Proxy prédit à l'avance que la requête va échouer. La requête ne devrait pas être transmise à d'autres localisations. Seul le SIP Proxy responsable de l'usager identifié par son URL peut répondre par cette classe de réponses.
   - **603 Decline** : signale que l'appelé est occupé ou qu'il ne désire pas répondre à ses appels.
   - **604 Does not exist Anywhere** : cette réponse indique que l'usager spécifié dans la requête n'existe pas.
   - **606 Not acceptable** : cette réponse peut être utilisée pour implanter une négociation. Cette réponse indique que certains aspects de la session ne sont pas acceptables par le UAS appelé.

2.4 Call Processing Language

Call Processing Language (CPL) a été accepté en tant que standard proposé de l'IETF en 2004. Il est conçu pour permettre aux utilisateurs de décrire et contrôler leurs propres services téléphoniques sur Internet. CPL permet aux utilisateurs de modifier et de supprimer leurs services à tout moment [17]. C'est un dialecte XML, indépendant des protocoles de signalisations. Il peut être exécuté sur SIP ou sur le protocole H.323 de l' ITU-T. Notons que dans ce travail, nous traitons seulement le protocole SIP. CPL est un langage qui ne permet pas l'utilisation de boucles ni de récursivité. Il induit de ce fait un potentiel d'erreurs dans les environnements hautement distribués comme l'Internet.

Il est en réalité conçu pour être sûr pour les utilisateurs non-expérimentés afin de décrire leurs politiques personnalisées.

Les scripts (politiques) CPL peuvent être exécutés sur le terminal de l'utilisateur ou sur le serveur proxy responsable de l'utilisateur. Une politique peut être représentée sous forme d'arbre de décision contenant des noeuds et des arrêtes (liens entre noeuds), où chaque noeud et lien correspondent a une balise XML dans le langage CPL. Un noeud (exemple <incoming>, <proxy>) signifie qu'il y a un évènement qui va se produire ou une action à prendre. Un lien (exemple <address is = "sip :bob@exemple.com">) spécifie quelle décision doit être prise. Une politique est exécutée si la requête est adressée à l'utilisateur ou l'auteur de la politique. Les conditions dans CPL sont basées sur un ensemble restreint d'évènements (incoming et outgoing) et de comportements prédéfinis, représentés par les actions (proxy, redirect et reject).

Les scripts CPL sont vérifiés off-line et ne causent pas de conflits dans le cas d'un seul utilisateur, car il y a un seul service qui va être exécuté lors d'un évènement. Suite au développement d'extensions de CPL (exemple Less, Accent - Appel) qui permettent l'expression des boucles dans leurs scripts et offrent plus de flexibilité aux usagers, il est éminent de vérifier l'apparition de nouveaux conflits et le cas échéant les résoudre.

Le sujet des interactions de fonctionnalités dans CPL a été étudié dans le mémoire de maitrise de Y. Xu [39] et dans l'article [40].

Xiaotao Wu est le premier chercheur qui a proposé l'idée d'étendre le langage CPL aux informations de présences et a publié sa conception dans l'Internet Draft de l'IETF [38]. Sa contribution dans le domaine de la téléphonie sur Internet porte sur la possibilité de collecter les informations de présences pour les utilisateurs des systèmes de services téléphoniques. Notre étude ne traite pas la présence, vu la complexité et le dynamisme du modèle.

Chapitre 3

État de l'art

3.1 Introduction

Plusieurs travaux de recherche ont été menés pour définir un cadre théorique adéquat à la modélisation des fonctionnalités et à la détection de leurs interactions. Ces études [21, 10, 37] développent des méthodes ayant pour but d'éviter les conflits et assurent le bon fonctionnement du système sur les réseaux.

Le probléme des interactions de fonctionnalités peut être traité en se basant sur une variété d'approches et de points de vue. Un certain nombre de travaux existe dans la littérature pour résoudre le problème de conflits entre fonctionnalités et politiques. Plusieurs auteurs ont conclu que certaines interactions indésirables résultent d'incohérance des spécifications. Dans ce sens, les rapports les plus importants peuvent être trouvés dans [9, 6] où des interactions de fonctionnalités sont modélisées sous forme d'incohérance entre spécifications de logique temporelle.

Calder et al. [5] mettent en évidence trois axes de recherche pour traiter les interactions de fonctionnalités, à savoir l'approche du génie logiciel, les méthodes de vérification formelles et les techniques on-line. Ces dernières sont destinées à détecter les interactions de fonctionnalités au moment de leur exécution, tandis que les deux précédentes interviennent au moment de la création du service.

Dans ce qui suit, nous détaillerons les principales approches de détection des interactions de fonctionnalités ou les conflits entre politiques du contrôle d'appels téléphoniques.

3.2 Les méthodes off-line

Les méthodes off-line sont des méthodes de détection des interactions au moment de la conception du système. Nous décrivons les plus importantes dans les prochaines sous-sections.

3.2.1 Méthodes du génie logiciel

Les méthodes du génie logiciel peuvent être exploitées pour réduire et éliminer les interactions de fonctionnalités indésirables avant leur implémentation. Les travaux, [14, 12] révèlent des architectures pour les services de télécommunications qui obligent les utilisateurs à créer des combinaisons *sûres* de fonctionnalités. L'intervention avec des mesures correctives à chaque étape de la création de services permet une meilleure gestion au niveau de la spécification des besoins, de l'analyse et l'implantation des systèmes, ainsi qu'au niveau du contrôle de la qualité des services offerts.

Les méthodes du génie logiciel éliminent les interactions pendant la phase de conception en modélisant d'une part différemment les fonctionnalités et politiques et d'autre part, en précisant à l'avance les comportements qui peuvent se produire suite à certaines combinaisons de fonctionnalités ou de politiques. Ainsi, résultera une solution idéale pour des interactions de fonctionnalités spécifiques. Mais le système devient éventuellement plus difficile à entretenir et à agrandir. Aussi, une seule solution parmi un ensemble de solutions possibles, sera proposée pour tous les utilisateurs.

3.2.2 Méthodes formelles

Une majeure partie des travaux, adressant le problème d'interaction de fonctionnalité, est fondée sur l'application des techniques formelles à la spécification et la vérification des systèmes et de leurs fonctionnalités.
Les méthodes formelles sont plus rigoureuses que les techniques de génie logiciel [5] [1]. Elles requièrent des notations spécialisées afin de traiter les cas à un niveau d'abstraction plus élevé.

[1] Bien que de nombreux auteurs considèrent que les méthodes formelles font partie des méthodes de génie logiciel, nous suivons dans ce travail l'idée soutenue dans l'article [5], où une distinction est faite entre ces deux méthodes.

Il existe deux types de modèles ; les modèles de propriétés et les modèles de comportements.

- Les modèles de propriétés vérifient si une propriété qui satisfait un système lors de l'exécution d'une seule fonctionnalité, satisfait encore le système en cas de rajout d'une ou plusieurs fonctionnalités ensemble.
- Les modèles de comportements analysent des combinaisons de deux ou plusieurs services et détectent les interactions en terme d'accessibilité "reachability", d'impasse "deadlock", de non-déterminisme ou d'incohérence logique [5].

Il y a aussi des méthodes qui combinent les deux modèles.

Selon [9], les fonctionnalités A et B sont en conflit si et seulement si le modèle de spécification réalisant leur conjonction A^B n'existe pas. La méthode de détection utilise le model-checker Cospan [11]. Une justification théorique à été donnée dans [1]. Le premier article classique sur ce sujet [5] évoque les conflits d'hypothèses en tant qu'une des causes principales des interactions de fonctionnalités.
D'autres travaux [13, 18] se sont basés sur l'idée que les interactions de fonctionnalités sont le résultat de l'exécution d'actions conflictuelles. Mais comment peut-on dire que deux actions sont en conflit ? Pour répondre à cette question, les auteurs de [37, 10], ont poussé l'analyse à une plus haute granularité en considérant les pré et post-conditions des actions. Ce point sera développé dans le reste de ce mémoire (voir chapitres 4 et 5)

3.2.3 La méthode de détection de Nakamura et al.

Nakamura et al. [21] définissent les interactions de fonctionnalités comme étant des avertissements sémantiques pour un ensemble de scripts. Ils définissent huit types d'avertissements. Ensuite, ils simulent un scénario pour tester si son comportement est similaire à l'un des avertissements. Nakamura et al. ont appliqué leur méthode sur le langage CPL et ont introduit l'appellation du "script CPL sémantiquement sûr". Un script est dit sémantiquement sûr si et seulement si le script ne satisfait aucun avertissement sémantique [21]. Dans ce qui suit, nous présentons les différents avertissements sémantiques.

Multiple forwarding addresses (MFAD) :

Lors de l'exécution d'un script, on peut atteindre une action où l'ensemble de localisation contient plus qu'une adresse. Nous citons un exemple classique a cet avertissement sémantique : "redériger les appels rentrants simultanément à la messagerie vocale et à mon cellulaire". L'appel n'atteint jamais mon cellulaire, donc il y a un conflit.

Unused subactions (USUB) :

L'utilisateur non-expérimenté peut exprimer une politique dont une partie est redondante ou non utilisable. Afin d'alléger la charge du réseau, l'avertissement sémantique USUB doit être éliminé.

Call rejection in all execution paths (CRAE) :

Cet avertissement apparaît lorsque les feuilles de l'arbre de décision (politique) ont la même valeur <reject>, c.-à-d., n'importe quel chemin dans l'arbre mène au même résultat.

Address set after address switch (ASAS) :

Une politique peut être contradictoire en elle même. Nous citons l'exemple : pour tout appel sortant si l'appelé est Bob alors rejeter la communication sinon établir une communication avec Bob.

Overlapping conditions in single switch (OCSS) :

Cet avertissement se manifeste lorsque suite à un choix entre deux conditions la satisfaction d'une condition1 implique la satisfaction de la condition2. Exemple de politique : pour tout appel rentrant, si le nom de l'appelant commence par la lettre "B" alors rejeter cet appel sinon si le nom de l'appelant commence par les lettres "Bo" alors rediriger cet appel à mon cellulaire. Dans ce cas, chaque fois que le nom de l'appelant commence par la lettre "B" la deuxième partie de la politique ne sera pas exécutée, donc elle est redondante.

Identical actions in single switch (IASS) :

Cet avertissement est un peu similaire à CRAE et se produit lorsqu'une même action est spécifiée pour toutes les conditions d'un même noeud interne (switch) de l'arbre de décision. Si une telle situation apparaît, alors le switch en question manque d'expressivité et il doit être éliminé afin de réduire la complexité de la logique.

Overlapping conditions in nested switches (OCNS) :

Une condition1 bloque l'exécution d'une condition2 pour le même switch. Exemple : pour tout appel rentrant, si le nom de l'appelant commence par les lettres "Bob" alors rejeter l'appel, sinon si le nom de l'appelant commence par la lettre "B" rediriger l'appel à mon cellulaire. Si Bob appelle, alors la branche de redirection ne sera jamais exécutée, donc elle est redondante.

Incompatible conditions in nested switches (ICNS) :

Cet avertissement sémantique apparait lorsque deux conditions imbriquées sont incompatibles. Exemple : pour un appel rentrant, si l'adresse de l'appelant est bob@exemple.com et si l'adresse de l'appelant est alice@exemple.com, alors acheminer l'appel à tom@exemple.com. L'appel ne sera jamais acheminé à tom@exemple.com puisque les deux adresses sont différentes et il n'est pas possible de recevoir un appel provenant de deux utilisateurs différents simultanément.

Il faut observer que ces avertissements sémantiques n'indiquent pas nécessairement des erreurs, mais plutôt des possibilités d'erreurs.

Détection des interactions basée sur la méthode de Nakamura et al.

La détection des conflits se fait entre deux scripts pouvant contenir des informations implicites ou imcomplètes. Pour ce fait, Nakamura et al. transforment les scripts originaux en scripts complets. Un script est complet si et seulement si, il n'existe aucun comportement par défaut pour tout chemin d'exécution possible (c.-à-d. toute action doit être spécifiée explicitement dans le corps du script). En appliquant l'algorithme suivant on obtient des scripts complets sans changer la logique du script original.

1. Ajouter à chaque condition son complémentaire (c.-à-d., une branche *sinon*).

2. Spécifier l'action appropriée à chaque feuille de l'arbre de décision (en particulier pour les branches *sinon*).
3. Ajouter un évènement declencheur à la politique, s'il n'en existe pas déjà (c.f., incoming ou outgoing call).

De cette façon, des scripts (ou comportements) obtenus sont déterministes. La combinaison de deux scripts complets génère un script complet. Basée sur l'idée des avertissements, la détection se fait en comparant le script résultant avec les avertissements. S'il n'y a aucun avertissement qui satisfait le script complet, on peut déduire qu'il est sémantiquement sûr et qu'il n'y a pas de conflits.

Discussions et leçons retenues

Nakamura et al. ont introduit des critères pour détecter huit classes de conflits. Cependant, il peut exister d'autres classes de conflits qui ne sont pas couvertes par les critères de Nakamura et al. Donc, le fait d'adopter cette méthode n'implique pas la détection de tous les conflits dans un langage.

Dans ce travail, nous nous inspirons de l'approche de Nakamura et al. Ainsi, nous définissons des comportements conflictuels pour valider les systèmes de politiques LESS et APPEL. Autrement dit, ces définitions seront utilisées pour détecter les interactions indésirables des systèmes.

3.2.4 La méthode de détection de Gorse et al.

Gorse et al., [10] décomposent les fonctionnalités en plusieurs parties descriptives. Chaque partie représente le comportement de la fonctionnalité pour un état spécifique du cycle de l'appel. Formellement, une fonctionnalité X est notée $\mathcal{F}_X$. Chaque partie descriptive d'une fonctionnalité $\mathcal{F}_X$ est notée $\mathcal{D}_{X,m}$, où X est la fonctionnalité et m est un entier identifiant le rang de la partie descriptive. La notation $\mathcal{F}_X = \{\mathcal{D}_{X,1}, \cdots, \mathcal{D}_{X,n}\}$ indique que la fonctionnalité $\mathcal{F}_X$ comporte n parties descriptives. Chaque partie descriptive est composée de quatre ensembles de propriétés nommés : pré-conditions, évènements déclencheurs, résultats et contraintes. Chacun de ces ensembles est constitué de propriétés élémentaires.

Nous citons ci-dessous des exemples de propriétés élémentaires :

1. $call(A, B)$ l'utilisateur A tente de communiquer avec l'utilisateur B.
2. $redirect(A, C)$ l'appel provenant de l'utilisateur A est transféré à l'utilisateur C.
3. $subs(A, CallForwarding)$ l'utilisateur A est abonné à la fonctionnalité $CallForwarding$.
4. $concerns(A, CallForwarding)$ indique que la fonctionnalité $CallForwarding$ s'applique à l'utilisateur A.

Nous introduisons dans ce qui suit les ensembles de propriétés mentionnés ci-dessus :

Ensemble des pré-conditions :

Décrit l'état du système avant l'activation de la fonctionnalité. Cet ensemble de propriétés est noté $\mathcal{P}_{X,m}$ et contient les informations concernant les utilisateurs abonnés à la fonctionnalité (c.f., : $subs(A, CallForwarding)$) et les utilisateurs influencés par la fonctionnalité (c.f., : $concerns(A, CallForwarding)$).

Ensemble des évènements déclencheurs :

Contient la ou les actions qui déclenchent la fonctionnalité et est noté $\mathcal{T}_{X,m}$.

Ensemble des résultats :

Exprime les résultats attendus ainsi que l'état du système après l'exécution de la fonctionnalité. Cet ensemble est noté $\mathcal{R}_{X,m}$.

Ensemble des contraintes :

Représente des restrictions supplémentaires sur les variables utilisées dans les ensembles de propriétés $\mathcal{P}_{X,m}$, $\mathcal{T}_{X,m}$ et $\mathcal{R}_{X,m}$. Cet ensemble est noté $\mathcal{C}_{X,m}$, avec $\mathcal{C}_{X,m} = \{(U_1, K_1, U_2), \cdots, (U_i, K_j, U_{i+1})\}$, où $i, j \geq 0$. $U_1, \cdots, U_i$ sont les variables utilisées dans $\mathcal{P}_{X,m}$, $\mathcal{T}_{X,m}$ et $\mathcal{R}_{X,m}$. $K_1, \cdots, K_j$ représentent l'égalité ($=$) ou l'inégalité ($\neq$) entre les variables.

Pour illustrer la méthode de détection des interactions de fonctionnalités de Gorse et al., considérons les deux fonctionnalités Originating Call Screening (OCS) et Call Forward Always (CFA) introduites dans le chapitre Introduction et motivation 1 (voir Fig. 1.1). Chaque fonctionnalité ($\mathcal{F}_{OCS}$ et $\mathcal{F}_{CFA}$) peut être représentée sous forme de quatre ensembles de propriétés comme le montre le paragraphe suivant.

$\mathcal{D}_{OCS,1}$ de Originating Call Scrining :

$$\begin{aligned}
\mathcal{P}_{OCS,1} &= \{subs(A, OCS), concerns(A, OCS), OCS_list(C)\} \\
\mathcal{T}_{OCS,1} &= \{call(A, C)\} \\
\mathcal{R}_{OCS,1} &= \{deny_call(A, C), call_denied(A, C)\} \\
\mathcal{C}_{OCS,1} &= \{(A \neq C)\}
\end{aligned}$$

$\mathcal{D}_{CFA,1}$ de Call Forward Always :

$$\begin{aligned}
\mathcal{P}_{CFA,1} &= \{subs(B, CFA), concerns(B, CFA), CFA_list(C)\} \\
\mathcal{T}_{CFA,1} &= \{call(A, B)\} \\
\mathcal{R}_{CFA,1} &= \{redirect(A, C), call(A, C)\} \\
\mathcal{C}_{CFA,1} &= \{(A \neq B), (A \neq C), (B \neq C)\}
\end{aligned}$$

Après la combinaison par jointure des deux fonctionnalités $|\mathcal{D}_{OCS,1} \wedge \mathcal{D}_{CFA,1}|$ on obtient les ensembles de propriétés suivants :

$$|\mathcal{D}_{OCS,1} \wedge \mathcal{D}_{CFA,1}| = \mathcal{D}_{OCS\_CFA,1}$$

$$\begin{aligned}
\mathcal{P}_{OCS_CFA,1} &= \{subs(A, OCS), concerns(A, OCS), OCS_list(C), subs(B, CFA), \\
&\quad concerns(B, CFA), CFA_list(C)\} \\
\mathcal{T}_{OCS_CFA,1} &= \{call(A, C), call(A, B)\} \\
\mathcal{R}_{OCS_CFA,1} &= \{\mathbf{deny_call(A, C)}, call_denied(A, C), redirect(A, C), \mathbf{call(A, C)}\}
\end{aligned}$$

$$\mathcal{C}_{OCS\_CFA,1} = \{(A \neq B), (A \neq C), (B \neq C)\}$$

Dans le même ensemble de résultats $\mathcal{R}_{OCS\_CFA,1}$ se trouve deux résultats contradictoires. $deny\_call(A, C)$ et $call(A, C)$, d'où le conflit par "incoherence transitive". Tous les autres types de conflits sont détectés de la même manière. Pour plus de détails consulter [10].

Cette méthode présuppose que les conflits entre actions comme celui qu'on vient de mentionner soient connus. Notre méthode effectue l'identification de ces conflits.

3.2.5 La méthode de détection de Wu et al.

Wu et al. [37] ont introduit l'algorithme de Merge et l'ont utilisé pour détecter les interactions de fonctionnalités du systeme LESS. Cette méthode consiste à décomposer les scripts LESS (ou arbres de décision) en un ensemble de régles de décision. Chaque régle de décision représente le chemin de la racine de l'arbre de decision à chaque feuille de l'arbre. Une régle de décision est composée des quatre entités suivantes : un évenement déclancheur, un ensemble d'actions, un ensemble de contraintes et l'adresse où chaque action sera exécutée.

Dans ce qui suit, nous décomposons, à titre d'exemple, les deux arbres de décisions des figures 1.2 et 1.3 présentés dans le chapitre 1.

Transférer tout appel provenant de mon patron sur mon cellulaire sinon l'accepter si l'appelant n'est pas mon patron.

Cette arbre est décomposable en deux régles de décisions : $\mathcal{R}_{1,1}$ et $\mathcal{R}_{1,2}$ telles que :

$$\begin{aligned}
\mathcal{R}_{1,1} &= \{incoming, transfer, \{string\text{-}switch\ caller = "Patron"\}, \\
&\quad \{address\text{-}switch\ sip\ :mon\text{-}cell@exemple.com\}\} \\
\mathcal{R}_{1,2} &= \{incoming, accept, \{string\text{-}switch\ caller \neq "Patron"\}, \\
&\quad \{address\text{-}switch\ sip\ :mon\text{-}tel@exemple.com\}\}
\end{aligned}$$

Renvoyer les appels en provenance de Alice à ma messagerie vocale, sinon accepter tout autre appel.

Décomposition de l'arbre.

$$\begin{aligned}\mathcal{R}_{2,1} &= \{incoming, redirect, \{string\text{-}switch\ caller = "Alice"\},\\ &\quad \{address\text{-}switch\ sip:messagerie@exemple.com\}\}\\ \mathcal{R}_{2,2} &= \{incoming, accept, \{string\text{-}switch\ caller \neq "Alice"\},\\ &\quad \{address\text{-}switch\ sip:mon\text{-}tel@exemple.com\}\}\end{aligned}$$

Ensuite, les régles doivent êtres normalisées, afin de grouper les différentes entités des régles ensemble. L'ordre n'intervient pas, sauf pour les conditions (Wu et al., ont imposé un ordre prioritaires). En effet, les conditions sont indépendantes entre elles, donc il n'y aura pas de changement par rapport à l'arbre de décision original. Dans notre cas, par contre, les régles sont normalisées, car on utilise une seule condition par script.

L'étape de la fusion consiste à combiner l'ensemble des régles normalisées en un seul ensemble nommé $\mathcal{S}$, comme le montre l'expression suivante (correspondante à la figure 1.4 arbre à gauche) :

$$\begin{aligned}\mathcal{S} &= \{\mathcal{R}_{1,1} \wedge \mathcal{R}_{1,2} \wedge \mathcal{R}_{2,2} \wedge \mathcal{R}_{2,2}\}\\ \mathcal{S} &= \{incoming, \{transfer, redirect, accept\}, \{string\text{-}switch\ caller = "Patron",\\ &\quad string\text{-}switch\ caller \neq "Patron", string\text{-}switch\ caller = "Alice",\\ &\quad string\text{-}switch\ caller \neq "Alice"\}, \{address\text{-}switch\ sip:mon\text{-}cell@exemple.com,\\ &\quad address\text{-}switch\ sip:messagerie@exemple.com,\\ &\quad address\text{-}switch\ sip:mon\text{-}tel@exemple.com\}\}\end{aligned}$$

Cette méthode nécessite la définition d'un tableau de conflits. Pour notre exemple on considère le tableau proposé par Wu et al. reproduit ici dans le tableau 3.1. On remarque bien que les deux actions `redirect` et `transfer` peuvent causer un conflit de

neutralisation (ligne 3, colonne 4 du tableau 3.1).

	accept	reject	redirect	transfer	merge	m-update	term	call
accept	A(m)	C	C	E	E	E	E	R
reject	C	A(r)	C	D	D	D	D	E
redirect	C	C	A(a)	D	D	D	D	E
transfer	E	-	-	A(a)	C	C	C	E
merge	R	-	-	C	-	C	C	R
m-update	R^1, E^1	-	-	C	C	A(m)	C	R^1, E^1
term	E	-	-	D	D	D	-	E
call	R	-	-	E	E	E	E	R

m-update : `media-update`, term : `terminate`, - : no interaction, C : action conflict
A(m) : attribute conflict on media, A(r) : attribute conflict on reason, E : enabling
A(a) : attribute conflict on adress, D : disabling conflict, R :resource competition
R^1 : `media-update` for unholding calls competes resources with `call` or `accept`
E^1 : `media-update` for holding calls enables `call` or `accept`

TAB. 3.1 – Call control action conflict table for handling `incoming` trigger (Wu et al. [37, Table 3])

Le tableau 3.1 montre les conflits entre les actions du système LESS. Wu et al. [37] supposent qu'un appel utilise par défaut l'audio comme media de communication et qu'il n'y a qu'un seul dispositif audio par terminal. Pour utiliser la vidéo et le texte, les utilisateurs peuvent manipuler plusieurs fenêtres vidéo et sessions de messages instantanés simultanément si leur CPU et la bande passante du réseau le permettent. Dans le tableau 3.1, on ne considère pas les cas de concurrence sur les ressources pour les conversations video et texte.

Le tableau 3.1 n'est pas symétrique. Dans ce tableau, les actions des lignes sont exécutées avant les actions des colonnes. Par exemple : ligne 1, colonne 7 veut dire que l'action `accept` sera exécutée en premier et par la suite l'action `terminate`. Le tableau 3.1 est élaboré pour les appels rentrants.

3.3 Les méthodes on-line

Les méthodes on-line effectuent la détection et la résolution des interactions au moment de l'exécution des services. Il existe deux classes de techniques on-line qui se dis-

tinguent par la localisation du contrôle.

- Les Feature Managers sont caractérisées par la centralisation des contrôleurs qui supervisent et contrôlent l'exécution des services. Les services impliqués ne se connaissent pas forcément.

- Les approches basées sur la Négociation exigent que les services coopèrent et communiquent entre eux directement par le biais d'un espace partagé de données publiques, ou à travers leurs agents qui utilisent des techniques distribuées d'intelligence artificielle. Pour cette classe, la logique de contrôle est dite distribuée.

La détection et la résolution des interactions en temps réel, signifient que des mesures correctives sont prises seulement lors de l'apparition de l'interaction. Cependant, cette approche gère un grand nombre de contraintes sur des fonctionnalités rarement utilisées. Ces méthodes sont peu applicables. Cela est dû à la difficulté de détecter et résoudre les interactions simultanément et en temps réel. D'autre part, cette méthode est non conforme aux lois de la protection des droits de propriété, puisque on doit connaitre les fonctionnalités auquelles sont abonnées les clients, pour étudier leurs interactions.

Dans ce mémoire, les méthodes on-line ne seront pas étudiées, Cependant, la détection des conflits entre actions que nous faisons peut être utile pour ces méthodes.

3.4 Autres méthodes

Pour complétude dans cette section, nous mentionnerons différentes visions du problème des interactions de fonctionnalités, qui ne seront pas utilisées dans ce travail.

Une classification qui a été utilisée dans [6] et qui a été reprise par plusieurs chercheurs est la classification basée sur les causes des interactions. Cette classification identifie dès le départ les cas problématiques et indique si des solutions existent aux conflits en question. Pour ce point de vue, trois sources principales du problème sont identifiées : les limites dues aux réseaux de communication, les problèmes intrinsèques aux systèmes répartis et les violations d'hypothèses.

La catégorisation par cycle de vie [7], quant à elle, analyse les interactions de fonctionnalités pendant un cycle de vie de l'application où elles peuvent être mieux contrôlées.

L'approche par configuration [6], procède en identifiant les entités logiques impliquées dans une interaction. Cameron et al. [6] différencient le nombre d'utilisateurs et le nombre de composants impliqués. Ils séparent aussi le terme "utilisateur" et "caractéristiques du terminal".

Le point de vue organisationnel [7], se concentre sur la responsabilité d'une interaction et de ses conséquences. Ainsi, le problème est réduit à une question de gestion d'interactions de fonctionnalités [12].

Amyot et al. [2] proposent une approche pour résoudre les conflits interactifs des services personnalisés pour la téléphonie sur Internet. Ils présentent une architecture combinant (1) la création de politiques pour les services de communication personnalisés, (2) la validation de ces services et (3) la prise en charge des conflits qui peuvent y exister.

D'autres techniques visent à empêcher les interactions de fonctionnalités via des contraintes architecturales. Les auteurs de [14] coordonnent les accès des fonctionnalités et politiques aux ressources à partager. Des messages jouent le rôle d'un jeton qui rend les actions des fonctionnalités sérialisables. L'organisation en séquence des fonctionnalités engendre un schéma de priorité. Avec cette approche, l'interaction peut échouer. Dans le cas contraire, elle implémente une seule stratégie de résolution qui est la précédence. Cette méthode manque donc de flexibilité, au niveau de la résolution.

Nous présentons dans ce qui suit une méthode de filtrage des conflits entre politiques basée sur les ontologies. Cette méthode a été appliquée au système Appel.

3.4.1 Méthodes de filtrage

Les méthodes de détection des interactions de fonctionnalités sont des méthodes complexes et profondes, d'une part, vue la nécessité de considérer toutes les combinaisons possibles des actions des politiques. D'autre part, les méthodes de détection des inter-

actions sont coûteuses en termes de temps d'exécution et d'espace mémoire. Aussi, il est impossible dans des cas de vérifier la totalité du système (on ne se contente que de spécifier un modèle abstrait, qui ne reflète pas nécessairement la globalité du système réel). Pour cette raison, les méthodes de filtrage sont souvent sollicitées pour faire des prédictions des éventuels conflits. Le filtrage est donc une étape qui précède la détection afin d'avertir les concepteurs des systèmes téléphoniques aux possibles conflits.

Campbell et al. [8] ont introduit une technique semi-automatique pour le filtrage des conflits entre les actions des politiques du contrôle d'appel téléphonique. Les auteurs ont intégré leur méthode de filtrage des conflits au système APPEL. Cette méthode est basée sur l'utilisation d'ontologies.

"Une ontologie est un ensemble de termes employés pour décrire et représenter un domaine de connaissance, ainsi que les relations logiques entre les termes du domaine " [22]. En d'autres termes, une ontologie est un vocabulaire commun pour partager les informations d'un domaine spécifique donné, incluant ses termes clés, leurs sémantiques et les règles d'inférence. Campbell et al. ont utilisé le langage OWL (Web Ontology Language [31]) pour définir les ontologies du système APPEL. OWL est basé sur le langage XML.

Selon [8], les conflits dans APPEL résultent d'incohérences entre les actions des politiques. Campbell et al. ont employé le concept d'*effet* et ont assigné à chaque action un ou plusieurs effets. Deux actions ayant le même effet et qui sont exécutées simultanément peuvent conduire à un conflit entre politiques. En outre, les actions ont des paramètres. Par exemple l'action `add-medium` peut avoir l'un des paramètres suivant ; *audio*, *vidéo* ou *tableau-blanc*. Les paramètres peuvent aussi influer sur la cohérence des actions des politiques [8]. Ces éléments sont décrits en utilisant le langage OWL.
Par exemple l'action `remove-medium`(*audio*) peut être considérée incohérente avec une deuxième action `remove-medium`(*audio*). Les auteurs estiment qu'il y a un potentiel de conflit si la première action retire le seul dispositif *audio*. À ce moment-là, la deuxième action ne peut pas s'exécuter, puisqu'il n'y a plus de dispositif audio utilisé pour la conversation téléphonique. Dans cette situation, l'utilisateur sera notifié du potentiel de conflit entre ses politiques.

La méthode de Campbell et al. analyse seulement les incohérences entre paires d'actions. Les incohérences entre trois actions ou plus ne sont pas considérées.

Les conflits potentiels entre les actions peuvent être déduits de l'ontologie définie et intégrée dans le noyau du système APPEL. La détection des conflits se fait en deux étapes en utilisant l'algorithme suivant :

- Premièrement, deux actions qui partagent au moins un effet commun sont identifiées comme conflit potentiel.
- Deuxièmement, les actions ayant des paramètres sont analysées. Lorsque deux actions disposent de la même valeur du paramètre, alors il y a possiblement un conflit entre ces actions.
- Sinon, on suppose qu'il n'y a pas de conflit.

Les auteurs font l'hypothèse que les conflits entre les actions sont symétriques et donc le nombre total des paires d'actions à analyser est $\frac{n*(n-1)}{2}$, où n est le nombre d'actions définies dans le système APPEL. La détection des conflits est commutative (si Action1 et Action2 sont en conflit alors Action2 et Action1 le sont aussi) et associative (toutes les combinaisons possibles des actions sont traitées).

Cette méthode a rendu possible une meilleure extensibilité du système APPEL. C.-à-d., à chaque fois qu'on rajoute de nouvelles actions au système APPEL, il suffit de rajouter leurs effets et paramètres sans se soucier des conflits qui peuvent en résulter. De plus, elle réduit considérablement la complexité du modèle et le temps de détection des possibilités de conflit entre politiques.

Nous pensons que cette méthode a deux inconvénients :

- Elle ne couvre pas la totalité des cas conflictuels puisque les auteurs considèrent juste $\frac{n*(n-1)}{2}$ cas, où n est le nombre d'actions à vérifier. Pourtant, dans la réalité, pour un système contenant n actions, il y a $n*n$ paires d'actions. Nous montrerons dans ce mémoire des exemples de conflits non-symétriques.
- L'ordre d'exécution des actions n'est pas pris en considération.

Cette technique sera discutée ultérieurement dans la section 5.3 du chapitre 5. Nous démontrons que ces deux points peuvent influer sur les résultats de la détections des conflits. Notre méthode de détection considère l'ordre d'exécution entre les actions et traite toutes les paires d'actions.

Dans ce mémoire, les autres méthodes discutées précédemment ne seront pas discutées ultérieurement. Notre contribution pourrait être utilisée dans leur cadre et pourrait faire l'objet de recherches ultérieures.

3.5 Discussion

Les politiques que nous prenons en considération peuvent être représentées sous forme d'arbre de décision (c.f., figures 1.2 et 1.3, contenant un événement déclencheur, un ensemble de conditions et une action). Selon [37], les événement déclencheurs (triggers) et les conditions ne causent pas de conflits. Nous présentons dans ce travail une méthode de détection des interactions de politiques qui se base sur la détection des interactions entre les actions des politiques impliquées.

Plusieurs travaux ont adopté les méthodes formelles pour l'analyse de services de communication. Parmi les plus significatifs, on note [21, 27, 40]. Notre méthode formelle à été conçue pour détecter les conflits dans le système Less [37] et les conflits du syteme Appel [24]. Nous avons combiné le modèle de propriétés et le modèle de comportements expliqués précédemment à la section 3.2.2. Plus de détails sur notre méthode peuvent être trouvés dans les chapitres 4 et 5.

Dans cette étude, nous cherchons à analyser formellement le fonctionnement des systèmes Less et Appel. Pour cela, nous commençons par la création d'un modèle abstrait des systèmes en question (une description des systèmes sous forme d'automates). Ensuite, nous définissons une liste de spécifications fonctionnelles (un ensemble de propriétés que les systèmes doivent satisfaire). Enfin, nous vérifions si les modèles des systèmes considérés satisfont les spécifications en utilisant la technique de vérification formelle du model-checking.

Chapitre 4

Premier cas d'étude : LESS et interactions dans LESS

Nous introduisons dans ce chapitre notre méthode de détection des conflits entre actions du systeme Less [37], en utilisant le langage formel de premier ordre Alloy [13].

4.1 Vue d'ensemble de l'architecture de LESS

Less est conçu spécifiquement pour créer des services exécutables sur les systèmes d'extrémité du réseau, c.-à-d. les terminaux comme les téléphones, les ordinateurs, les télécopieurs, les répondeurs, etc. Less est une extension du Call Processing Language (CPL) (voir section 2.4 du chapitre 2). Il adopte une structure arborescente pour représenter les services de télécommunication et permet aux utilisateurs inexpérimentés de créer leurs propres services qui roulent sur des terminaux de la téléphonie sur Internet. Une application a été développée afin de manipuler Less, appelée Columbia University Telecommunication service Editor (CUTE)[36]. Nous citons un exemple de service afin de comprendre le fonctionnement de Less.

La figure 4.1 exprime une règle de Less générée par l'outil de déscription CUTE. Une règle de décision est exprimée par CUTE sous forme d'arbre binaire. La racine de l'arbre décrit le type d'appel que l'utilisateur veut établir. Dans notre cas, c'est un appel rentrant. La date (heure et jour) est la première condition à vérifier. Si on se situe dans l'intervalle de temps entre 08 h 30 min et 12 h 30 min le jeudi 12 Octobre 2006, alors on procède à la condition suivante, sinon on répond à l'appel. Suite à l'information de

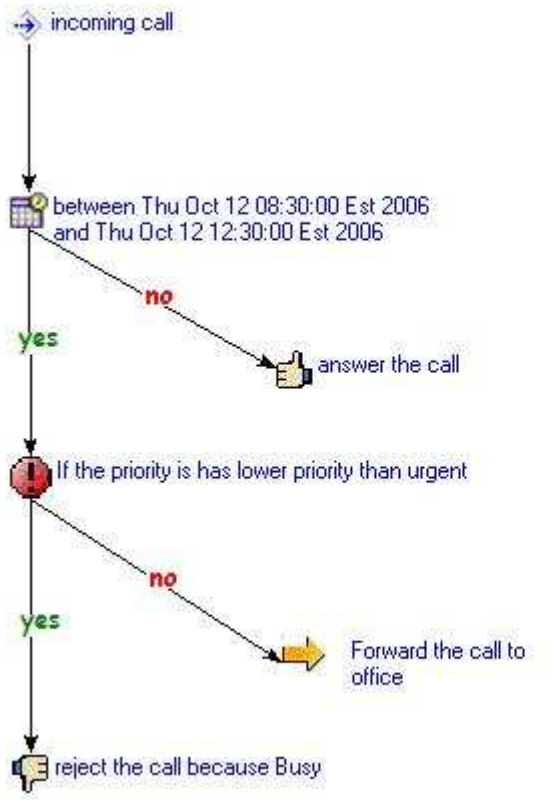

FIG. 4.1 – Règle de décision

priorité de la communication émise par l'appelant, l'appelé décide de rejeter l'appel en cas de priorité inférieure à urgent. Autrement, il transfère l'appel à son bureau.

4.2 Les éléments de LESS

Toute règle de décision dans LESS est constituée des quatre éléments suivants :

Les événements déclencheurs

Un événement déclencheur (ou trigger) est une entité qui invoque un script (ou une règle). Un script LESS est invoqué si, et seulement si, son événement déclencheur est exécuté. L'événement déclencheur doit être la racine de l'arbre de décision. Il ne peut y avoir plus qu'un dans une même règle. Un événement déclencheur ne peut aussi avoir

qu'un seul successeur. Le successeur d'un événement déclencheur est une condition.

L'événement déclencheur est un point d'entrée à n'importe quel service. Il a la même signification que le `toplevelaction` défini en CPL. LESS exige trois types d'événements déclencheurs ; `incoming` (appel rentrant, voir figure 4.1), `outgoing` (appel sortant) et `timer` (un événement qui se déclenche à une date et une heure prédéfinies par l'auteur du script).

Les conditions

Une condition est une exigence à respecter pour exécuter une règle. Seulement, les conditions d'un script donné peuvent diriger les décisions prises suite à un appel téléphonique. Une condition doit être le successeur d'un événement déclencheur ou d'une autre condition. Une condition ne peut pas avoir plus que deux successeurs. Exemple : la figure 4.1 a une condition de priorité de l'appel et une condition de date et heure.

Les actions

Les actions influent sur le contexte de l'appel et changent le statut de la communication. Seulement, les actions peuvent être les feuilles d'un arbre de décision, voir figure 4.1 où les actions sont : accept (answer the call), transfer (forward the call) et reject. Une règle peut contenir plusieurs actions qui se succèdent. Un événement déclencheur ne succède jamais à une action.

Les modificateurs

Un modificateur ne peut être que l'élément parent d'une action. Il spécifie l'adresse à laquelle l'action doit être executée. Par exemple : le modificateur de l'action `accept` ne peut pas être une adresse différente de celle du terminal qui héberge la politique contenant l'action `accept`. Tandis que le modificateur de l'action `transfer` doit être différent de l'adresse de l'utilisateur qui héberge la politique contenant l'action `transfer`.

4.3 Les actions dans LESS

LESS permet l'exécution de plusieurs actions séquentiellement ou simultanément. LESS traite deux types d'actions, les `signaling actions` et les `non-signaling actions`.

Ces dernières, sont nommées `log`, `wait` et `mail` et ne causent pas de conflit avec les `signaling actions` [37]. C'est pour cette raison qu'on ne traite pas leurs interactions. Dans ce travail, on se concentre plus sur les `signaling actions` présentées dans ce qui suit.

4.3.1 Accept

L'action `accept` permet à l'utilisateur d'accepter un appel rentrant. L'action `accept` ne dispose pas de paramètre, mais un modificateur est nécessaire. Le modificateur `location` par exemple, est souvent utilisé avec l'action `accept` pour préciser la localisation du terminal de l'appelé. Après l'exécution de `accept`, l'état de l'appel sera "*la session est établie*".

4.3.2 Reject

L'action `reject` impose au serveur SIP de rejeter toute tentative d'établir une session de conversation. L'action `reject` termine immédiatement l'exécution du script LESS. Donc les états de l'appel et du terminal en question ne changent pas. C'est la raison pour la quelle il y a deux arguments pour notifier l'utilisateur, à savoir "status" ou "reason". L'argument "status" est obligatoire et peut prendre l'une des quatre valeurs suivantes :

- busy : la ligne est occupée ;
- notfound : partie appelante non trouvable ;
- reject : appel refusé ;
- error : erreur interne du serveur.

L'argument "reason" est optionnel et permet au script de divulguer à la partie appelante la raison du rejet.

4.3.3 Redirect

L'action `redirect` force le serveur SIP à rediriger la partie appelante à un ensemble d'adresses prédéfinies par la partie appelée. L'action `redirect` termine immédiatement l'exécution du script LESS. Ainsi les états de la session et du terminal de l'appelé ne

changent pas. Le seul paramètre de l'action `redirect` est "permanent". Il signale le résultat spécifié par l'auteur de la règle. Sa valeur par défaut est "no" pour indiquer qu'il n y a pas de redirection permanente. Si l'utilisateur spécifie dans ses règles une nouvelle adresse, alors la valeur de ce paramètre est "yes".

4.3.4 Call

L'action `call` permet à l'agent utilisateur d'effectuer un appel sortant. L'action `call` possède un seul paramètre "Timeout", qui désigne la durée de temps allouée à une tentative d'appel.
Suite à l'exécution de l'action `call`, cinq situations peuvent se produire :

- accepted, si l'appel est accepté ;
- redirected, si l'appel est redirigé vers une autre adresse ou messagerie ;
- busy, si l'appelé est occupé et ne répond pas ;
- noanswer, si l'appelé ne répond pas pendant le temps "timeout" ;
- failure, si l'appel échou pour n'importe quelle autre raison.

4.3.5 Transfer

L'action `transfer` force l'agent utilisateur à transférer les appels existants à une autre adresse. L'action `transfer` requiert les mêmes paramètres que l'action `call`.

4.3.6 Media-update

L'action `media-update` incite l'agent utilisateur à mettre à jour les paramétres de la session en cours. L'action `media-update` ne requiert pas de paramètre ; elle est utilisée pour traiter les services de mi-appels, par exemple "mute" et "hold".

4.3.7 Merge

L'action `merge` fusionne les appels en une conférence entre différents usagers. Par défaut, le flux audio sera combiné et envoyé à tous les participants de la conférence. Le flux vidéo sera expédié à tous les utilisateurs impliqués. S'il y a des sessions mises en attente (wait) ou en mode muet (mute), leurs statuts restent inchangeables.

L'action `merge` possède deux paramètres, "uri" et "subject" qui spécifient comment sélectionner les appels pour les fusionner.

- Si le paramètre "uri" est présenté, tous les appels impliquant l'une des valeurs de "uri" seront fusionnés.
- Si le paramètre "subject" est présenté, tous les appels ayant l'une des valeurs de "subject" seront fusionnés.

Si d'autres utilisateurs veulent participer à une conférence existante, ils peuvent employer n'importe quelle valeur dans la liste de "subject" ou de "uri" pour se référer à la conférence en question.

4.3.8 Terminate

L'action `terminate` permet à l'agent utilisateur de terminer une ou plusieurs sessions de conversation et tentatives d'appel. Il y a trois paramètres pour cette action qui spécifient comment terminer les sessions de conversation.

- Si le paramètre "uri" est présenté, tous les appels à une des valeurs de "uri" seront terminés.
- Si le paramètre "subject" est présenté, tous les appels qui ont une des valeurs du champ "subject" seront terminés.
- Si le paramètre "calls" a la valeur "all", tous les appels qui coïncident avec une valeur du champs "subject" et "uri" seront terminés.
- Si le paramètre "calls" a la valeur "last", la dernière session qui a été établie sera terminée.
- Si le paramètre "calls" a la valeur "this", les appels associés à l'événement déclencheur en cours seront terminés.

4.4 Détection des intéractions de fonctionnalités dans LESS

Si deux règles causent un conflit, alors la cause peut être l'une des situations suivantes :

- l'une des règles contient un conflit en elle même ;
- les deux règles contiennent des conflits par conception ;
- la combinaison des deux régles cause le conflit.

Soient R_1 et R_2, deux règles. Leur conjonction engendre une règle $\mathcal{R}$. Dans ce travail, on suppose toujours que la déctection des conflits se fait pour $\mathcal{R}$. Cela veut dire que si $\mathcal{R}$ contient un conflit, alors les règles R_1 et R_2 sont en conflit, puisque $\mathcal{R} = \mid R_1 \wedge R_2 \mid$.

Notre méthode de détection se base sur les hypothéses suivantes :

- chaque action a une pré-condition. Si cette pré-condition n'est pas satisfaite, l'action ne peut pas être executée ;
- chaque action a une post-condition. Après l'execution d'une action, l'état du systéme change ;
- une règle peut contenir plusieurs actions. Cependant, suite à la détection du premier couple d'actions conflictuelles dans une règle, la recherche de conflit s'arrête (c.-à-d. les cas de conflit entre plus que deux actions ne sont pas traités) ;
- si deux actions sont en conflit, alors la ou les politiques qui invoquent ces actions sont en conflit.

Pour vérifier les interactions entre deux actions, nous avons définit un ordre d'exécution. Il est nécessaire de tester les différentes possibilités de conflits dans des ordres différents. Une situation conflictuelle peut avoir lieu entre deux actions A1 et A2, si les post-conditions de A1 ne sont pas cohérentes avec les pré-conditions de A2. Tous les cas possibles seront détaillés dans ce qui suit.
Wu et al. [37] définissent quatre types de situations conflictuelles, à savoir : *conflit d'action*, *conflit d'attribut*, *conflit de concurrence sur les ressources* et *conflit de neutralisation*. Ils définissent aussi un type d'*interactions permises*. Ces situations ont été

traduites sous formes de prédicats, exprimant le comportement du conflit en question. Par la suite, on détermine les instances qui vérifient leurs propriétés.
Le tableau 4.1 décrit les pré et post-conditions des `signaling actions` et il est la base de la vérification. Dans ce travail, nous considérons le tableau 4.1 comme "fait" et vrai tout au long de l'exécution des règles.

ACTIONS	**pré-condition**	**Post-conditions**	
		État de l'appel	État du terminal
Accept	incoming call setup pending, media devices available	call setup finalized, a session is setup	media devices occupied
Reject	incoming call setup pending	call setup finalized	no change
Redirect	incoming call setup pending	call setup finalized	no change
Call	media devices available	a session is setup	media devices occupied
Transfer	one or more sessions, media devices occupied	all sessions terminated	media devices available
Media-update	one or more sessions, media devices occupied	all sessions alive	media transmission changed
Merge	one or more sessions, media devices occupied	all sessions merged	media devices occupied
Terminate	one or more sessions, media devices occupied	all sessions terminated	media devices available

TAB. 4.1 – The context assumption and expected result of call control actions (Wu et al. [37, Table 2])

4.4.1 Conflit d'action

Le *conflit d'action* concerne les post-conditions des actions. C'est un conflit entre deux actions, si ces dernières mènent au même état après leur exécution, ou si une `Action1` produit une post-condition qui n'est pas compatible avec une des post-conditions d'une `Action2`. C.-à-d. :

– Supposons que `Action1` produit une post-condition X et que `Action2` produit la même post-condition X. On suppose que si on est dans un état X on ne peut pas y rester après l'execution d'une autre action, car chaque action change l'état du systeme après son exécution.

– ou si `Action1` produit une post-condition Y et `Action2` produit une post-condition Z, telles que Y et Z ne sont pas compatibles, alors il y a un *conflit d'action*.

Nous proposons dans le tableau 4.2, les incompatibilités conduisant à un *conflit d'action*.

Les combinaisons d'incompatibilité *Conflit d'action*	
Post-condition *(Action1)*	Post-condition *(Action2)*
call setup finalized	call setup finalized
all sessions terminated	all sessions merged
all sessions terminated	all sessions alive
all sessions terminated	all sessions terminated
all sessions merged	all sessions terminated
all sessions merged	all sessions alive
all sessions alive	all sessions terminated
all sessions alive	all sessions merged

TAB. 4.2 – Ensemble des incompatibilités du prédicat *conflit d'action*

La détection des conflits s'effectue dans une même règle, voir figure 4.1. Un utilisateur peut générer une règle ayant plusieurs actions.

Dans une première partie de ce travail, on essaie de générer des règles sémantiquement justes. Prenons l'exemple des actions, `accept` et `redirect` (voir tableau 4.1), exécutées successivement dans l'ordre présenté. Après l'exécution de l'action `accept`, l'appel est établi et l'état de l'appel sera call setup finalized. Ensuite, en exécutant l'action `redirect`, on reproduit le même état d'appel qui a été déjà atteint, alors il y a *conflit d'action*.

Nous traitons de la même manière le couple d'actions `transfer` et `merge` (voir tableau 4.1). Après l'exécution de l'action `transfer`, la session est terminée pour l'appelé et l'etat de l'appel sera all sessions terminated. Ensuite, en exécutant l'action `merge`, l'état d'appel all sessions merged est produit. Cette dernière post-condition représente une mise-à-jour des paramètres de l'appel (voir tableau 4.2). Or, l'appel a déjà été transféré et la session est achevée. Ce qui implique l'existence d'un *conflit d'action*.

Nous présentons dans le tableau 4.3 les résultas de la détection pour le *conflit d'action* à l'aide de l'outil Alloy Analyzer.

ACTIONS	Accept	Reject	Redirect	Transfer	Merge	Media-update	Terminate	Call
Accept		C	C					
Reject	C		C					
Redirect	C	C						
Transfer					C	C	C	
Merge				C		C	C	
Media-update				C	C		C	
Terminate				***C***	***C***	***C***		
Call								

C : Conflit d'action

TAB. 4.3 – Vérification du prédicat *Conflit d'action*

Ces résultats affirment les propositions avancées par Wu et al. [37] de conflits possibles. D'autre part, la vérification a révélé de nouveaux cas conflictuels entre les couples d'actions (`terminate` et `transfer`), (`terminate` et `merge`) et (`terminate` et `Media-update`). Les actions `terminate` et `transfer` mènent au même état de l'appel, qui est all sessions terminated (voir tableau 4.1 et tableau 4.2). Il y a donc un *conflit d'action*.

Soit le couple d'actions `terminate` et `merge`, exécutées successivement dans l'ordre présenté. Après l'exécution de l'action `terminate`, l'état de l'appel change à "toutes les sessions sont terminées" all sessions terminated. Suite à l'exécution de l'action `merge`, on génère un état d'appel all sessions merged qui est incompatible avec celui atteint précédemment (voir tableau 4.2). D'où le *conflit d'action*.

Après l'exécution de l'action `terminate`, l'état de l'appel sera all sessions terminated. D'autre part, l'exécution de l'action `Media-update` génère un état d'appel all sessions alive qui est incompatible avec celui atteint précédemment (voir tableau 4.2). D'où le *conflit d'action*.

4.4.2 Conflit d'attribut

Un conflit d'attribut se manifeste lors de l'exécution de deux actions ayant le même nom mais des attributs différents. LESS considère les modificateurs comme attributs

pour les actions. Deux actions ayant le même nom causent un conflit d'attribut si leurs modificateurs sont différents.

ACTIONS	Accept	Reject	Redirect	Transfer	Merge	Media-update	Terminate	Call
Accept	A							
Reject		A						
Redirect			A					
Transfer				A				
Merge					***A***			
Media-update						A		
Terminate							***A***	
Call								***A***

A : Conflit d'attribut

TAB. 4.4 – Vérification du prédicat *Conflit d'attribut*

Dans le tableau 4.4, nous présentons les résultats de la détection pour le prédicat *conflit d'attribut*. Ces résultats confirment les propositions de la détection fournies par Wu et al. ; aussi, nous avons détecté des nouveaux cas de conflit aprés avoir exécuter récursivement l'une des actions suivantes : `Merge`, `terminate` et `call`. Ici, il s'agit de rechercher des possibilités de cas conflictuels. C'est à l'auteur de la règle de juger si c'est un conflit ou non.

Wu et al. distinguent trois types de conflits d'attribut : *conflits d'attribut sur les medias*, *conflits d'attribut par raison* et *conflits d'attribut par adresse*. Voir plus de détails dans [37].

4.4.3 Conflit de concurrence sur les ressources

Ce conflit est basé en premier lieu sur la disponibilité du terminal au cours de l'appel. Deux actions peuvent être en concurrence sur les ressources. Les cas d'incompatibilités possibles, conduisant à un *conflit de concurrence sur les ressources*, sont proposés dans le tableau 4.5 .

Deux catégories d'incompatibilités menant au *conflit de concurrence sur les ressources* sont distinguées :

Les combinaisons d'incompatibilité du *Conflit de concurrence sur les ressources*	
Post-condition *(Action1)*	Pré-condition *(Action2)*
call setup finalized	media devices available
a session is setup	media devices available
Post-condition *(Action1)*	Post-condition *(Action2)*
a session is setup	call setup finalized
a session is setup	a session is setup
all sessions merged	a session is setup
all sessions alive	a session is setup

TAB. 4.5 – Ensemble des incompatibilités du prédicat *conflit de concurrence sur les ressources*

– incompatibilité entre l'état de l'appel de l'`Action1` et la pré-condition de l'`Action2`. Exemple : l'action `accept` produit l'état de l'appel a session is setup. La session est établie et le terminal est occupé. À l'exécution de l'action `call`, l'état du terminal contraint qu'il soit libre media devices available, d'où l'incompatibilité entre les deux actions (voir tableau 4.5). Cette situation engendre un *conflit de concurrence sur les ressources*, tableau 4.6 ;

– incompatibilité entre l'état de l'appel de l'`Action1` et l'état de l'appel de l'`Action2`. Exemple : l'action `merge` génère un état d'appel all sessions merged. Tous les appels sont fusionnés. En second lieu, l'exécution de l'action `accept` produit l'état de l'appel a session is setup alors que la session n'est pas encore terminée. L'action `accept` contredit l'état de l'appel de l'action `merge` (voir tableau 4.5) et génère un *conflit de concurrence sur les ressources*, tableau 4.6.

Le tableau 4.6 confirme les résultats du travail [37]. Alloy Analyzer ne détecte pas de nouveaux cas conflictuels.

4.4.4 Conflit de neutralisation

Deux actions causent un *conflit de neutralisation* si elles satisfont les deux propriétés suivantes :

ACTIONS	Accept	Reject	Redirect	Transfer	Merge	Media-update	Terminate	Call
Accept								R
Reject								
Redirect								
Transfer								
Merge	R							R
Media-update	R							R
Terminate								
Call	R							R

R : Conflit de concurrence sur les ressources

TAB. 4.6 – Vérification du prédicat *Conflit de concurrence sur les ressources*

1. Si `Action1` a X comme état de l'appel (post-condition) et `Action2` a Y comme pré-condition et (X, Y) est dans l'ensemble des incompatibilités du *conflit de neutralisation*;
   ET
2. Si l'état du terminal (post-condition) après l'exécution de l'`Action1` est incompatible avec une des pré-conditions de l'`Action2`.

Nous proposons dans le tableau 4.7 les incompatibilités menant à un *conflit de neutralisation*.

Les combinaisons d'incompatibilité du *Conflit de neutralisation*	
Post-condition_État de l'appel *(Action1)*	Pré-condition *(Action2)*
call setup finalized	one or more sessions
all sessions terminated	one or more sessions
ET	
Post-condition_État du terminal*(Action1)*	Post-condition *(Action2)*
no change	media devices occupied
media devices available	media devices occupied

TAB. 4.7 – Ensemble des incompatibilités du prédicat *conflit de neutralisation*

Prenons l'exemple du couple d'actions `accept` et `transfer` (voir tableau 4.1) exécutées successivement dans l'ordre présenté. Après l'exécution de l'action `accept`, l'appel est établi. L'état de l'appel sera call setup finalized. L'action `transfer` requiert la

pré-condition one or more sessions, c.-à-d. qu'il y a au moins une session en cours. On remarque déjà qu'il y a une incompatibilité. Pour trancher, il faut vérifier si l'état du terminal pour les deux actions est incompatible (voir tableau 4.7). Dans ce cas, l'état du terminal occupé est égal à media devices occupied pour les deux actions. Il n'y a donc pas de *conflit de neutralisation*.
Par contre, le couple d'actions `terminate` et `transfer` génère un *conflit de neutralisation*. En effet, l'état de l'appel de l'action `terminate` et une des pré-conditions de l'action `transfer`, respectivement all sessions terminated et one or more sessions, sont incompatibles et figurent dans l'ensemble des incompatibilités du conflit de neutralisation (voir tableau 4.7). De plus, l'état du terminal de l'action `terminate` est incompatible avec les pré-conditions de l'action `transfer`, respectivement media devices available et media devices occupied. Donc, il y a une intéraction de type *conflit de neutralisation*.
Le tableau 4.8 montre les résultats de la détection obtenus à l'aide de l'outil Alloy Analyzer.

ACTIONS	Accept	Reject	Redirect	Transfer	Merge	Media-update	Terminate	Call
Accept								
Reject				N	N	N	N	
Redirect				N	N	N	N	
Transfer					***N***	***N***	***N***	
Merge								
Media-update								
Terminate				N	N	N		
Call								

N : Conflit de neutralisation

TAB. 4.8 – Vérification du prédicat *Conflit de neutralisation*

Le tableau 4.8 confirme les résultats obtenus dans le travail de Wu et al. [37]. D'autre part, la vérification a révélé de nouveaux cas conflictuels entre les couples d'actions (`transfer` et `merge`), (`transfer` et `Media-update`) et (`transfer` et `terminate`).

Soit le couple d'actions `transfer` et `merge` (voir tableau 4.1), exécutées successivement dans l'ordre présenté. Après l'exécution de l'action `transfer`, l'état de l'appel change à toute session terminée all sessions terminated qui est incompatible avec la pré-condition one or more sessions de l'action `merge` (voir tableau 4.7). De plus, l'action `transfer`, change l'état d'appel a media devices available qui est incompatible avec la

pré-condition media devices occupied de l'action `merge` (voir tableau 4.7). D'où le *conflit de neutralisation*.

Après l'exécution de l'action `transfer`, l'état de l'appel sera all sessions terminated, qui est incompatible avec la pré-condition de l'action `Media-update`. De plus, l'action `transfer` génère l'état du terminal media devices available qui est contradictoire avec la pré-condition media devices occupied de l'action `Media-update`, one or more sessions. Les actions `transfer` et `Media-update` engendrent un *conflit de neutralisation*.
C'est de la même manière que le comportement de *conflit de neutralisation* entre les deux actions `transfer` et `terminate` exécutées successivement est detécté.

4.4.5 Interactions permises

Ces interactions sont des comportements désirables et ne sont pas des anomalies dans le système. Les interactions permises produisent le cas contraire du conflit de neutralisation.

Soient `Action2` et `Action2` exécutées successivement. Si les post-conditions(état de l'appel et état du terminal), de l'`Action1` impliquent les pré-conditions de l'`Action2`, alors il n y a pas de conflit et l'interaction est permise.

En vue d'illustrer un cas d'*interaction permise*, nous proposons l'exemple du couple d'actions `accept` et `transfer` exécutées dans l'ordre présenté. Les post-conditions en *état d'appel* de l'action `accept`, sont égales à "call setup finalized" et "a session is setup". Ces dernières sont cohérentes avec la pré-condition "one or more sessions" de l'action `transfer`.

Aussi, l'état du terminal après l'exécution de l'action `accept` sera "media devices occupied", qui indique que le terminal est occupé. Également, l'action `transfer` exige que l'état du terminal soit occupé "media devices occupied" pour s'exécuter. Par conséquent, il y a une *interaction permise* entre les actions `accept` et `transfer`.

Le prédicat *Interactions permises* exprime les comportements désirables dans le système LESS. Les résultats de la détection des *interactions permises* sont présentés dans le tableau 4.9.

ACTIONS	Accept	Reject	Redirect	Transfer	Merge	Media-update	Terminate	Call
Accept				P	P	P	P	
Reject								P
Redirect								P
Transfer	P							P
Merge								
Media-update	P							P
Terminate	P							P
Call				P	P	P	P	

P : Interactions permises

TAB. 4.9 – Vérification du prédicat *Interactions permises*

Nous ne détectons pas de nouveaux cas d'*interactions permises* par rapport au travail de Wu et al. [37].

4.5 Discussion

Le tableau 4.10 récapitule les résultats des interactions dans LESS. Ces résultats ont été détaillés précédemment dans la section 4.4. Les cases colorées représentent les nouveaux cas conflictuels. Notre méthode a permis de détecter neuf nouvelles interactions entre règles exprimées par LESS.

ACTIONS	Accept	Reject	Redirect	Transfer	Merge	Media-update	Terminate	Call
Accept	A	C	C	P	P	P	P	R
Reject	C	A	C	N	N	N	N	P
Redirect	C	C	A	N	N	N	N	P
Transfer	P			A	C, ***N***	C, ***N***	C, ***N***	P
Merge	R			C	***A***	C	C	R
Media-update	P, R			C	C	A	C	P, R
Terminate	P			N, ***C***	N, ***C***	N, ***C***	***A***	P
Call	R			P	P	P	P	R, ***A***

C : Conflit d'action, A : Conflit d'attribut, N : Conflit de neutralisation
R : Conflit de concurrence sur les ressources, P : Interactions permises

TAB. 4.10 – Récapitulatif des résultats de la vérification du systeme LESS

Le tableau 4.10 présente des cases vides. Ces cases ne fournissent aucun jugement sur la cohérence entre les actions ; aucune interaction n'est alors détectée. Par exemple, si les actions `terminate` et `reject` sont exécutées dans l'ordre, notre méthode ne détecte aucune interaction. Ce qui concorde avec l'analyse de Wu et al.[37].

Prenons l'exemple des actions `transfer` et `reject` exécutées dans l'ordre. Si on transfère un appel rentrant, cela veut dire qu'on a dépassé l'étape de mi-appel. L'action `transfer` termine la communication et génère un état d'appel "all sessions terminated". Par conséquent, on ne peut pas exécuter une autre action qui exige que l'état de l'appel soit au début du cycle de l'appel "incoming call setup pending". Donc, l'action `reject` ne peut jamais succéder l'action `transfer`. Le système LESS, par conception, ne permet pas la combinaison de ces actions. Dans l'analyse des interactions présentée dans le travail [37], ces questions d'ordre admissible des actions sont traitées par une analyse manuelle. Elles pourraient être traitées en utilisant la notion de l'état de l'appel utilisée dans l'analyse du système APPEL.

Comme le but de ce chapitre est de valider le plus possible de résultas de [37], cette notion ne sera pas introduite ici.

Notre méthode de détection résoud la totalité des cas conflictuels du modèle abstrait du système Less.

Chapitre 5

Deuxième cas d'étude : APPEL et interactions dans APPEL

5.1 Le langage de politiques APPEL

APPEL (ACCENT Project Policy Environment/Language) est un langage de programmation de services. Il a été conçu pour exprimer les politiques sur des terminaux distants (téléphones, répondeurs, etc.). Le langage est défini dans le rapport technique [26] et une description de son application au domaine du contrôle d'appels téléphoniques est donnée dans [29]. APPEL est supporté par les protocoles de gestion de session SIP et H.323.

APPEL est conforme au modèle ECA (Event-Condition-Action) dans sa description des politiques. C'est-à-dire que les politiques du système APPEL sont invoquées à la suite d'événements déclencheurs (appelés aussi `triggers`), quand certaines conditions sont vraies et engendrent l'exécution de certaines actions. Pour cette étude de cas, nous considérons deux types d'événements déclencheurs, à savoir `OutgoingCall` pour les appels sortants et `IncomingCall` pour les appels rentrants.

Lorsqu'un événement déclencheur est exécuté, suite à un appel rentrant ou suite à l'ajout d'un nouveau participant à la communication en cours, le serveur responsable de la session identifie toutes les politiques applicables. Ces dernières sont les politiques de l'appelant, celles de l'appelé ainsi que les politiques de haut niveau (les politiques des

organisations. Par exemple : "on ne reçoit plus d'appel aux bureaux en dehors des heures de travail et tous les appels seront redirigés vers l'agent de garde"). Les politiques de haut niveau sont plus prioritaires que les autres politiques. L'applicabilité des politiques dépend de la description des conditions et de leur période de validité. Le résultat sera un ensemble d'actions. Les événements déclencheurs et les conditions peuvent être combinés par des operateurs logiques.

Le langage APPEL est basé sur la présentation XML, les politiques ne peuvent alors pas être définies par un utilisateur ordinaire (ou non-expert de XML). Pour aider les utilisateurs ordinaires à exprimer leurs politiques, le système APPEL est doté d'un assistant de politiques convivial (un "Wizard") qui permet aux usagers d'éditer et de créer leurs politiques en utilisant un langage quasi-naturel.

Malgré que le langage APPEL ait été développé en premier lieu pour contrôler les appels téléphoniques, il a été appliqué dans d'autres domaines. Par exemple, il a été utilisé pour la gestion des politiques reliées aux "soins à domiciles" ainsi qu'aux "réseaux de capteurs". Ce large choix d'applications est possible avec APPEL, car il est muni d'un langage noyau qui supporte les extensions aux *applications métiers*.

5.2 Pré et Post-conditions des actions dans APPEL

Notre méthode de détection se base sur l'analyse des pré et post-conditions des actions pour déterminer l'incohérence entre les politiques du système APPEL. Les pré-conditions d'une action indiquent les états du système afin que l'action puisse être exécutée. Les post-conditions d'une action spécifient les états résultants du système après l'exécution de l'action en question.

Deux actions peuvent être exécutées simultanément ou séquentiellement c'est à dire :

- Exécution simultanée : une action commence son exécution avant qu'une autre n'ait mis fin.
- Exécution séquentielle : une action commence son exécution à la fin de l'exécution d'une autre (c.-à-d. une action précède une autre).

Si deux actions ont des pré-conditions incohérentes alors leur exécution ne peut pas s'effectuer simultanément. De plus, deux actions qui s'exécutent successivement (l'une après l'autre), peuvent être en contradiction si l'une produit une post-condition incohérente avec une des pré-conditions de l'action suivante.

Nous détaillons les relations entre les pré et post-conditions des actions du système APPEL dans la section 5.3.

Le système APPEL englobe seize actions. Ce nombre d'actions est assez grand pour rendre le système difficile à analyser avec l'outil de vérification formelle Alloy Analyzer. Pour cette raison, notre modèle abstrait ne prendra en considération que les actions suffisantes et nécessaires à un appel téléphonique, tout au long du cycle de vie d'une communication.

Nous étudierons dans ce travail les dix actions du système APPEL suivantes :

- **connect_to** : Initier une session pour un appel sortant. Cette action ne peut être utilisée qu'avec l'événement déclencheur `OutgoingCall`.
- **reject_call** : Rejeter un appel rentrant. L'action reject_call n'est compatible qu'avec l'événement déclencheur `IncomingCall`.
- **forward_to** : Cette action permet de rediriger l'appel à une autre destination et ne peut être combinée qu'avec l'événement déclencheur `IncomingCall`.
- **fork_to** : Diriger l'appel à plusieurs adresses simultanément.
- **add_party** : Ajouter un participant à la communication déjà en cours (conférence).
- **remove_party** : Retirer un participant de la communication déjà en cours.
- **add_medium** : Ajouter un media (audio, video ou texte) à la communication en cours.
- **remove_medium** : Retirer un media de la communication en cours.
- **remove_default** : Permetre la libération d'un medium (audio, video ou texte).
- **disconnect** : Cette action termine une conversation en cours.

Bref, en APPEL une session téléphonique est amorcée par une connexion. L'usager peut rejeter ou transférer l'appel, ou il peut le rediriger à plusieurs adresses. Des usagers peuvent être ajoutés ou retirés. Des media peuvent être ajoutés ou retirés : par exemple,

une session qui commence en audio peut continuer en vidéo. Au début d'une session on lui affecte le medium "default" qui est normalement l'audio. La session se termine par une déconnexion.

Certaines actions, telle que connect_to, réservent implicitement le medium par défaut pour la communication (généralement l'audio). Les actions fork_to, add_party, remove_party, add_medium, remove_medium, remove_default et disconnect peuvent être combinées avec les deux événements déclencheurs `IncomingCall` et `OutgoingCall`.

Comme dans l'étude de LESS, notre approche de détection des conflits entre actions se base sur l'analyse des pré et post-conditions des actions du système Appel. Nous définissons dans le tableau 5.1 les pré et post-conditions des actions qui sont considérées dans cette étude. Le tableau 5.1 présente une perspective simplifiée et abstraite du processus de l'appel téléphonique pour le système Appel. L'élaboration de ce tableau est une tâche délicate car elle a une influence sur les résultats de la détection des conflits entre les actions.

ACTIONS	**Pré-condition**		**Post-conditions**	
	État de l'appel	État du terminal	État de l'appel	État du terminal
connect_to	NoCall	DefaultAvailable	CallSetup	DefaultReserved
reject_call	CallSetup	DefaultReserved	NoCall	DefaultAvailable
forward_to	CallSetup	DefaultReserved	CallForwarded	DefaultAvailable
fork_to	CallSetup	DefaultReserved	CallForked	DefaultReserved
add_party	MidCall	DefaultAvailable	PartyAddedToCall, MidCall	DefaultReserved
remove_party	MidCall, PartyAddedToCall	DefaultReserved	MidCall	DefaultAvailable
add_medium	MidCall	MediumAvailable	MidCall	MediumReserved
remove_medium	MidCall	MediumReserved	MidCall	MediumAvailable
remove_default	MidCall	DefaultReserved	MidCall	DefaultAvailable
disconnect	MidCall	DefaultReserved	NoCall	DefaultAvailable

Tab. 5.1 – Pré et post-conditions des actions dans APPEL

Pour comprendre ce tableau, prenons l'exemple de l'action `connect_to`. Le système Appel permet d'établir une communication téléphonique entre deux utilisateurs en exé-

cutant l'action `connect_to`. Pour que la connexion entre ces deux utilisateurs réussisse, les pré et post-conditions de l'action `connect_to` doivent être satisfaites, comme suit :

- L'appelant ne doit pas être impliqué dans une communication déjà en cours, comme l'indique la pré-condition NoCall.
  Et
- L'état du terminal de l'appelant doit être libre afin de lui permettre de joindre l'appelé. Ceci se traduit par la pré-condition état du terminal égale à DefaultAvailable.

Après l'exécution de l'action `connect_to`, l'état de la communication téléphonique change, comme l'indique le tableau 5.1. D'une part, l'état de l'appel change à CallSetup c.-à-d. la session est en train d'être établie, et d'une autre part, l'état du terminal sera changé à DefaultReserved, c.-à-d. le dispositif téléphonique est occupé.

Les incohérences entre pré et post-conditions pour trois actions et plus sont possibles. Toutefois, nous ne traiterons pas ces cas, pour des raisons de complexité du modèle. Dans ce travail, nous nous limiterons aux cas de conflits entre deux actions. Ce qui revient à analyser les actions deux par deux. Nous aborderons dans la section 5.3 les conflits entre actions en général. Puis, dans les sections 5.4, 5.5 et 5.6, nous détaillerons les cas conflictuels pour le système APPEL.

5.3 Ordre et Conflits entre actions du système APPEL

Nous avons mentionné précédemment que nous étudierons les dix actions de notre modèle abstrait du système APPEL. Nous analysons chaque combinaison possible des actions. Nous prenons l'exemple des deux actions `remove_medium` et `disconnect` exécutées dans cet ordre. L'action `remove_medium` libère le media et par la suite l'action `disconnect` termine la conversation déjà en cours. L'exécution successive des actions `remove_medium` et `disconnect` ne cause pas de conflits.

Prenons maintenant le cas où l'on exécute en premier l'action `disconnect` et subséquemment l'action `remove_medium`. Après l'exécution de `disconnect`, l'état de l'appel sera terminé et l'état du terminal sera libre. Donc, la communication est interrompue

et il n'y a plus de conversation. Tandis que, l'action `remove_medium` a besoin d'un état d'appel en cours d'exécution et d'un état du media réservé pour être exécutée. Il y a donc un conflit potentiel entre les actions `disconnect` et `remove_medium` exécutées dans cet ordre.

L'ordre d'exécution des actions est pris en considération dans notre méthode de détection des conflits. Cependant, pour les dix actions que nous analysons, il y a cent combinaisons possibles. Par rapport à notre modèle abstrait, notre méthode de détection des conflits couvre la totalité des cas pouvant être mis à exécution par des utilisateurs ordinaires.

La détection des conflits entre chaque couple d'actions se base sur les relations entre les pré et post-conditions des actions. Nous développerons dans ce qui suit les directives qui justifient nos choix des conflits.

1. Relations entre les pré-conditions de A et les pré-conditions de B :

   - La conjonction des pré-conditions des actions A et B est toujours vraie. Ces deux actions peuvent toujours être exécutées simultanément.
   - La conjonction des pré-conditions des actions A et B est validable. Dans certaines situations les actions A et B peuvent être exécutées simultanément.
   - Les pré-conditions de A et B ne sont pas simultanément validables. Donc les deux actions A et B ne peuvent pas être exécutées simultanément.
     Par exemple si les deux actions A et B utilisent la seule ressource disponible ou si elles ne peuvent s'exécuter qu'à des différentes phases du cycle de vie de la communication (les actions `connect_to` et `disconnect` illustrent bien cette situation).

2. Relations entre les post-conditions de l'action A et les pré-conditions de l'action B :

   - La conjonction des post-conditions de A et des pré-conditions de B est toujours vraie, alors B peut toujours être exécutée après que la première action A se termine.

- Les post-conditions de A et les pré-conditions de B sont simultanément validables. Alors, B ne peut être exécutée après A que dans certains cas. Par ailleurs, si une post-condition de A implique la pré-condition de B, alors les actions A et B peuvent être exécutées successivement.
- Les post-conditions de A ne sont pas simultanément validables avec les pré-conditions de B. Par conséquent, B ne peut pas être exécutée à la suite de A. Par exemple, si A libère une ressource audio dont B a besoin pour son exécution, alors A neutralise B. Ainsi, nous identifions le conflit de neutralisation. C'est le cas où A produit un état du système incohérent avec ce que l'action B requiert comme pré-conditions pour être exécutée.

3. Relation entre les post-conditions des actions A et B :

- Les post-conditions de A et B sont vraies. L'état du système après l'exécution de A et B est cohérent.
- Les post-conditions de A et B sont simultanément validables. Le résultat de l'exécution de A et B peut être cohérent dans certains cas.
- Les post-conditions de A et B ne sont pas simultanément validables. Par conséquent, l'état du système résultant sera incohérent après l'exécution de A et B. Par exemple, si A termine la session en cours, tandis que B continue la communication, alors le système atteint un état incohérent (dit aussi état impossible). Dans ce cas, A et B causent un conflit de résultat.

Pour ce travail, nous ne détaillons pas plus de problèmes, car cela rend la tâche de la détection plus compliquée. Nous nous limitons aux trois catégories de conflits entre actions suivantes :

- Conflit de simultanéité : deux actions ont des pré-conditions incohérentes et ne peuvent pas être exécutées dans la même phase de l'appel.
- Conflit de neutralisation : une action génère un état du système incohérent avec l'action qui la suit.
- Conflit de résultat : deux actions génèrent un état incohérent (ou impossible) du système, donc elles ne peuvent pas être exécutées simultanément.

Les deux aspects des pré et post-conditions qui vont être considérés sont l'état de l'appel et l'état du terminal. Dans ce travail, nous considérons les six cas suivants :

1. Conflit de simultanéité - état de l'appel.
2. Conflit de simultanéité - état du terminal.
3. Conflit de neutralisation - état de l'appel.
4. Conflit de neutralisation - état du terminal.
5. Conflit de résultat - état de l'appel.
6. Conflit de résultat - état du terminal.

Dans ce qui suit, nous aborderons le problème de l'identification de ces conflits.

5.4 Conflit de simultanéité

Il peut y avoir un *conflit de simultanéité* lorsque deux actions ne peuvent pas s'exécuter en même temps (ou en parallèle) au cours de l'appel téléphonique. C.-à-d. si deux actions ne sont pas exécutées au même état de l'appel ou au même état du terminal en pré-condition, alors elles ne doivent pas être exécutées dans la même phase de l'appel.

Nous définissons une phase comme étant une période de temps où l'état du système ne change pas. Si l'état du système change, alors nous passons à une nouvelle phase. La succession des pré et post-conditions des actions dans le temps produit les phases de l'appel.

Pour détecter un *conflit de simultanéité*, nous inspectons les pré-conditions des actions afin de vérifier leurs cohérences et décider s'il existe un conflit ou non. Ainsi, la figure 5.1 montre deux actions en exécution simultanée, où l'exécution débute dans la même phase de l'appel.

On suppose que `Action1` et `Action2` soient exécutées dans cet ordre. `Action1` et `Action2` ne seront pas en conflit de simultanéité si `Action2` est exécutée avant qu'`Action1` ne prenne effet, et que les pré-conditions des deux actions soient compatibles. Le cas

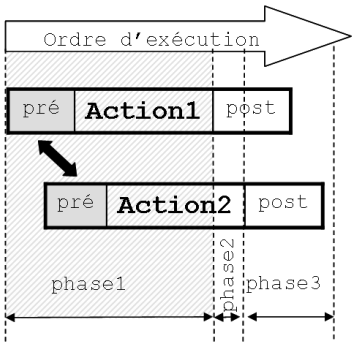

FIG. 5.1 – Conflit de simultanéité

contraire sera un comportement de *conflit de simultanéité*. L'exécution simultanée de deux actions peut générer un comportement conflictuel lorsqu'au cours d'une phase de l'appel, les actions possèdent des pré-conditions incohérentes.

Donc, le *conflit de simultanéité* est basé sur l'idée d'analyser les pré-conditions des actions. Toutefois, nous avons considéré dans le tableau 5.1 (des pré et post-conditions des actions), deux types de pré-conditions : état de l'appel et état du terminal. Nous distinguons alors deux éventualités reliées à ce type de conflit. La première éventualité concerne l'état de l'appel, alors que la seconde concerne l'état du terminal. Nous détaillerons dans ce qui suit ces deux éventualités.

5.4.1 Conflit de simultanéité - état de l'appel

Comme mentionné précédemment, pour ce conflit, nous cherchons à identifier tous les couples d'actions qui ne peuvent pas être exécutées au même état de l'appel du système. Notre approche requièrt l'identification d'un ensemble d'états incompatibles prédéfinis pour détecter les conflits entre les actions du langage APPEL. Nous proposons dans le tableau 5.2 les états incompatibles pour le *conflit de simultanéité - état de l'appel.*

Par exemple l'action `connect_to` ne peut pas être exécutée en même temps avec n'importe quelle autre action, puisqu'elle est la seule action qui doit être exécutée avant que la communication ne soit établie.

Les combinaisons d'incompatibilité *conflit de simultanéité - état de l'appel*	
Pré-condition *(Action1)*	Pré-condition *(Action2)*
MidCall	NoCall
MidCall	CallSetup
NoCall	MidCall
NoCall	CallSetup
CallSetup	MidCall
CallSetup	NoCall

TAB. 5.2 – Ensemble des incompatibilités du prédicat *Conflit de simultanéité - état de l'appel*

Prenons le cas des actions `connect_to` et `reject_call` exécutées dans l'ordre présenté (voir tableau 5.1). L'action `connect_to` exige qu'il n'y ait pas de communication déjà en cours pour qu'elle s'exécute avec succès. Tandis que, l'action `reject_call` a besoin de l'état de l'appel établi pour être exécutée. Étant donné que les deux pré-conditions ne peuvent pas être satisfaites en même temps alors, nous supposons que les pré-conditions "NoCall" et "CallSetup" sont incompatibles (voir tableau 5.2). Par conséquent, l'exécution simultanée des deux actions `connect_to` et `reject_call` dans cet ordre est conflictuelle.

Le tableau 5.8 représente tous les *conflits de simultanéité - état de l'appel* détectés en se basant sur l'ensemble des incompatibilités présenté dans le tableau 5.2.

5.4.2 Conflit de simultanéité - état du terminal

Ce conflit est similaire au conflit de simultanéité - état de l'appel, sauf qu'il considère l'état du terminal. Au cours d'une communication, les actions partagent les ressources du système téléphonique, ce qui peut générer des conflits liés à l'incohérence des pré-conditions au cours d'une phase de l'appel.

En effet, si une Action1 exige que l'état du terminal soit libre "DefaultAvailable" pour être exécutée et que l'Action2 requiert que le terminal soit occupé "DefaultReserved" pour être exécutée, alors la conjonction de l'Action1 et de l'Action2 présuppose un

Les combinaisons d'incompatibilité *conflit de simultanéité - état du terminal*	
Pré-condition *(Action1)*	Pré-condition *(Action2)*
DefaultAvailable	DefaultReserved
DefaultReserved	DefaultAvailable
MediumAvailable	MediumReserved
MediumReserved	MediumAvailable

TAB. 5.3 – Ensemble des incompatibilités du prédicat *Conflit de simultanéité - état du terminal*

état impossible du système et elle engendre un *conflit de simultanéité - état du terminal.*

Nous proposons dans le tableau 5.3 les pré-conditions incompatibles pour le *Conflit de simultanéité - état du terminal.*

Pour illustrer le *conflit de simultanéité - état du terminal*, nous citons l'exemple où les actions `add_party` et `remove_party` sont exécutées simultanément. L'action `add_party` exige que l'état du terminal soit occupé " DefaultReserved ", tandis que l'action `remove_party` exige que l'état du terminal soit libre " DefaultAvailable ". Les deux actions ne peuvent donc pas être exécutées simultanément durant une même phase de l'appel. Par conséquent, il y a un *conflit de simultanéité - état du terminal* entre les actions `add_party` et `remove_party`.

5.5 Conflit de neutralisation

Il est possible qu'une action donne lieu à un état du système, faisant en sorte qu'une autre action soit impossible à exécuter. Soient `Action1` et `Action2` exécutées dans l'ordre. Il peut y avoir un conflit entre elles, lorsque `Action1` neutralise `Action2`. L'ordre d'exécution des actions permet de déterminer les incohérences entre les post-conditions de `Action1` vis-à-vis des pré-condition de `Action2` (voir figure 5.2).

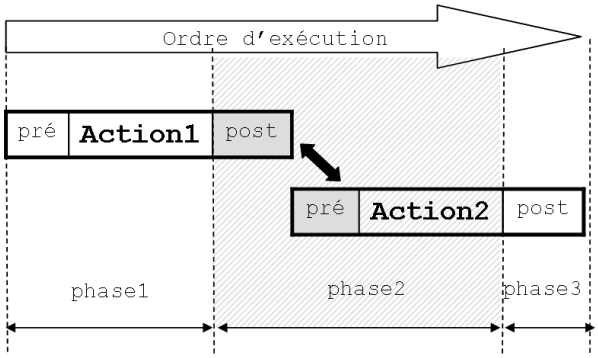

FIG. 5.2 – Conflit de neutralisation

Les conflits de neutralisation peuvent se produire entre deux actions dans le cas où la première action termine son exécution avant que la deuxième action n'entame son exécution.

Nous considérons qu'il n'y a pas de conflit de neutralisation entre Action1 et Action2 si leurs post et pré-conditions respectives sont cohérentes (c.-à-d. dans ce cas, qu'elles sont coïncidentes ou consécutives).

Les sous-sections 5.5.1 et 5.5.2 seront consacrées respectivement à détailler les conflits de neutralisations de types état d'appel et état du terminal.

5.5.1 Conflit de neutralisation - état de l'appel

Soient `Action1` et `Action2` exécutées dans l'ordre présenté. Supposons que `Action2` soit exécutée après que `Action1` n'ait terminé. Si `Action1` termine l'appel et que `Action2` exige qu'une communication soit en cours, alors ces deux actions ne peuvent pas s'exécuter successivement. Par conséquent, `Action1` neutralise `Action2` en raison de l'état de l'appel.

Les états de l'appel peuvent être incohérents, s'ils sont utilisés dans un ordre non approprié au cycle de vie d'un appel téléphonique (voir figure 5.3).

Le cycle de vie d'un appel téléphonique est la succession logique d'états de l'appel. Dans la figure 5.3, nous présentons un modèle abstrait du cycle vie d'un appel téléphonique constitué des trois états suivants : "NoCall", "CallSetup" et "MidCall".

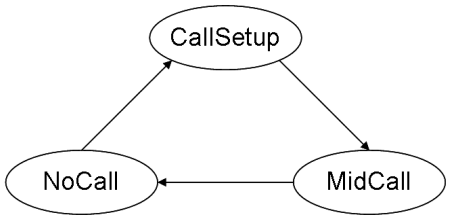

FIG. 5.3 – Cycle de vie d'un appel téléphonique

Les combinaisons d'incompatibilité *Conflit de neutralisation - état de l'appel*	
Post-condition *(Action1)*	Pré-condition *(Action2)*
NoCall	MidCall
CallSetup	NoCall
MidCall	CallSetup

TAB. 5.4 – Ensemble des incompatibilités du prédicat *Conflit de neutralisation - état de l'appel*

La figure 5.3 exprime un graphe dont les états représentent les pré et post-conditions et les arêtes représentent les actions. Si le séquencement des états du cycle de vie échoue, alors l'appel sera interrompu, par conséquent nous signalons l'existence d'un conflit.

Nous présentons dans le tableau 5.4 les incompatibilités possibles entre les post-conditions et les pré-conditions des actions pour le *conflit de neutralisation - état d'appel*.

L'exemple suivant illustre un cas de *conflit de neutralisation - état d'appel*. Soient les actions `disconnect` et `add_party` exécutées successivement. Après que l'action `disconnect` ne s'exécute et mette fin à la communication, l'état de l'appel sera "NoCall". Alors que l'action `add_party` exige que la session soit en cours pour qu'elle puisse s'executer. La post-condition "NoCall" de l'action `disconnect` est incohérente avec la pré-condition "MidCall" de l'action `add_party`. Il y a donc un *conflit de neutralisation - état d'appel* entre `disconnect` et `add_party`.

Les combinaisons d'incompatibilité *Conflit de neutralisation - état du terminal*	
Post-condition *(Action1)*	Pré-condition *(Action2)*
DefaultReserved	DefaultAvailable
DefaultAvailable	DefaultReserved
MediumAvailable	MediumReserved
MediumReserved	MediumAvailable

TAB. 5.5 – Ensemble des incompatibilités du prédicat *Conflit de neutralisation - état du terminal*

5.5.2 Conflit de neutralisation - état du terminal

S'il n'y a pas de conflit de neutralisation - état d'appel entre deux actions, on vérifie s'il y a une concurrence sur les ressources. Il peut y avoir un conflit si une action réserve le dispositif audio et que l'action suivante exige que le dispositif audio soit libre pour s'exécuter.

Le tableau 5.5 montre l'ensemble des incompatibilités entre les post-conditions et les pré-conditions des actions pour le *conflit de neutralisation - état du terminal*.

Pour clarifier le *conflit de neutralisation - état du terminal*, nous examinons le cas des deux actions `reject_call` et `fork_to` exécutées successivement. L'action `reject_call` produit une post-condition en état du terminal égale à "DefaultAvailable", c.-à-d. que le dispositif audio sera libre après l'exécution de cette action. Mais l'action `fork_to` exige que le dispositif audio soit occupé pour qu'elle puisse s'executer. Par consequent, les deux actions `reject_call` et `fork_to` ne peuvent pas être exécutées pendant la même phase de l'appel. On dit alors qu'il y a une possibilité de *conflit de neutralisation - état du terminal* entre `reject_call` et `fork_to`.

Les résultats de la détection du conflit de neutralisation pour toutes les actions sont présentés dans le tableau récapitulatif (TAB. 5.8) et seront discutés dans la section 5.7.

5.6 Conflit de résultat

Deux actions peuvent être en conflit si elles conduisent à deux états incohérents. Nous désignons ce type de conflit comme conflit de résultat, car l'état du système résultant, suite à l'exécution de deux actions, sera notre indicateur des possibilités de conflits. Nous analysons les post-conditions des actions pour ce type conflit. Autrement-dit, si deux actions génèrent des post-conditions incohérentes pendant la même phase de l'appel, alors il y a un conflit de résultat.

La figure 5.4 exprime une éventualité à considérer entre deux actions parmi plusieurs situations de concurrence entre actions, ainsi que l'ordre de leurs exécutions.

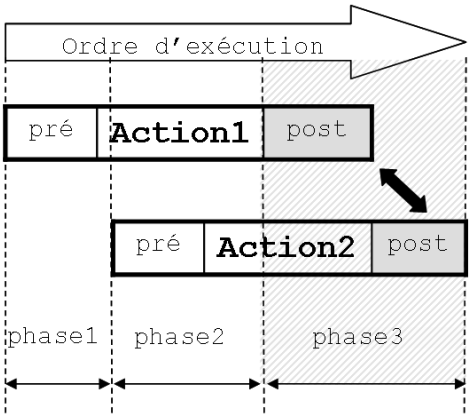

FIG. 5.4 – Conflit de résultat

Nous présenterons respectivement, dans les deux sous-sections suivantes le conflit de résultat relié à l'état de l'appel et à l'état du terminal .

5.6.1 Conflit de résultat - état de l'appel

Il peut y avoir un *conflit de résultat - état d'appel* entre deux actions si leurs post-conditions en état d'appel sont incohérentes. Nous distinguons les deux cas suivants :

1. Deux actions génèrent deux états d'appel qui ne respectent pas le cycle de vie de l'appel téléphonique (voir figure 5.3).

2. Les post-conditions résultantes, suite à l'exécution de deux actions, sont disjointes. Dans ce cas, le système atteint un état impossible. Par exemple, l'exécution simultanée des actions `reject_call` et `forward_to` conduirait à un état impossible. En effet, après l'exécution de l'action `reject_call`, la session est terminée et l'état de l'appel est "NoCall". De plus, l'action `forward_to` produit une post-condition en état d'appel "CallForwarded" sans qu'il y ait une indication sur l'achèvement de la session. L'action `forward_to` ne change pas l'état de la communication. A titre d'exemple, considérons la politique suivante, où l'action `forward_to` est utilisée lorsque la communication est en cours :

   "Renvoyer tous les appels rentrants à la messagerie vocale si l'utilisateur est occupé".

   Dans cet exemple, après l'exécution de `forward_to`, on ne peut pas savoir si l'appel est terminé ou s'il est en cours.

   Par conséquent, les post-conditions "NoCall" et "CallForwarded" sont disjointes et peuvent causer un *conflit de résultat - état d'appel* entre les actions `reject_call` et `forward_to`.

Le tableau 5.6 représente l'ensemble des incompatibilités des post-conditions pour le *conflit de résultat - état d'appel*.

En vue de mettre au clair le *conflit de résultat - état de l'appel*, relié au cycle de vie, citons l'exemple des deux actions `disconnect` et `add_party` exécutées dans l'ordre présenté. Après l'exécution de l'action `disconnect` la communication est terminée, et la post-condition est égale à "NoCall". La deuxième action `add_party` devra suivre l'ordre chronologique du cycle de vie de l'appel qui est "CallSetup". Par contre, l'action `add_party` génère la post-condition "MidCall", ce qui indique un saut d'un état du cycle de vie de l'appel. Ainsi, il y a un *conflit de résultat - état d'appel* entre les actions `disconnect` et `add_party`.

Les combinaisons d'incompatibilité *Conflit de résultat - état de l'appel*	
Post-condition *(Action1)*	Post-condition *(Action2)*
MidCall	CallSetup
NoCall	MidCall
CallSetup	NoCall
NoCall	CallForwarded
CallForwarded	NoCall

TAB. 5.6 – Ensemble des incompatibilités du prédicat *Conflit de résultat - état de l'appel*

Les combinaisons d'incompatibilité *Conflit de résultat - état du terminal*	
Pré-condition *(Action1)*	Pré-condition *(Action2)*
DefaultAvailable	DefaultReserved
DefaultReserved	DefaultAvailable
MediumAvailable	MediumReserved
MediumReserved	MediumAvailable

TAB. 5.7 – Ensemble des incompatibilités du prédicat *Conflit de résultat - état du terminal*

5.6.2 Conflit de résultat - état du terminal

Si `Action1` et `Action2` sont exécutées dans l'ordre présenté et qu'elles génèrent des post-conditions en état du terminal incompatibles pour la même phase, alors il y a un *conflit de résultat - état du terminal.*

Nous proposons dans le tableau 5.7 l'ensemble des incompatibilités possibles des post-conditions des actions dans APPEL pour le *conflit de résultat - état du terminal.*

Soient les actions `add_party` et `remove_default` exécutées dans l'ordre présenté. Si l'on applique le prédicat du *conflit de résultat - état d'appel* à ce couple d'actions, alors on ne détecte pas de conflit. Effectivement, les post-conditions en état d'appel de `add_party` et de `remove_default` sont compatibles. Par contre, il y a un conflit entre

ces deux actions.

En effet, après l'exécution de `add_party`, l'état du terminal indique que le dispositif audio est réservé "DefaultReserved" et que la communication est en cours. Ensuite, l'action `remove_default` libère le dispositif audio en produisant la post-condition "DefaultAvailable". Cependant, les actions `add_party` et `remove_default` ne doivent pas s'exécuter pendant la même phase, parce qu'il est impossible de satisfaire les deux post-conditions "DefaultReserved" et "DefaultAvailable" pendant la même phase de l'appel. C'est un état impossible à atteindre. Ainsi, il y a un *conflit de résultat - état du terminal* entre les actions `add_party` et `remove_default`.

Les résultats de la détection de tous les conflits su système APPEL sont présentés dans le tableau 5.8.

5.7 Discussion des résultats expérimentaux

Le tableau 5.8 représente les résultats de la détection des conflits pour le système APPEL. Nous avon discuté dans les sections 5.4, 5.5 et 5.6 comment aboutir à ces résultats. Notre méthode de détection à été appliquée sur un modèle abstrait du système APPEL, qui est constitué de dix actions (voir Annexe B). Chaque combinaison de deux actions a été vérifiée par six type de conflit.

Finalement, nous avons effectué six-cent tests, qui ont permis la détection de deux-cent-cinq conflits possibles. Il y a certains conflits triviaux (par exemple les actions `disconnect` et `add_party` exécutées successivement). Par contre, il y a des couples d'actions qui nécessitent une étude plus approfondie pour détecter les éventuels conflits (l'exemple du couple d'action `connect_to` et `reject_call` exécutées simultanément, énnoncé dans la sous-section 5.4.1).

Puisque l'analyse de tous les cas de concurrence entre les actions du système APPEL est une tache complexe, nous nous sommes limités à étudier seulement trois cas. Nous rappelons que tous les conflits détectés dans cette étude ne se réfèrent qu'à trois cas de concurrence précisés et résumés dans les figures 5.1, 5.2 et 5.4. Par conséquent, les conflits détectés par notre méthode ne sont plus fondés si le contexte change.

De quelle façon un expert du domaine de contrôle d'appel (ou du système APPEL) peut-il interpréter ces résultats ? En général un expert est guidé par une compréhension pragmatique du comportement du système. Par contre, notre méthode est formelle et a un haut niveau d'abstraction. Comme l'avons nous déjà mentionné précédemment, notre méthode de détection des conflits est basée sur l'analyse des pré-conditions et des post-conditions des actions. Cependant, les pré et post-conditions ne sont pas assez précises. En effet, il est impossible de savoir quels et combien de participants ou media sont ajoutés ou libérés. Par conséquent, la méthode proposée dans ce travail ne détecte les conflits mentionnés que pour certaines combinaisons des pré et post-conditions des actions.

Puisque l'idée de ce travail est d'identifier les paires d'actions qui présentent un grand potentiel de conflits, cette approche est efficace.

5.8 Conditions des politiques

Une extension de notre méthode offre aussi la possibilité de détecter les conflits en prenant en considération les conditions des politiques. Nous citons l'exemple des deux politiques suivantes :

- Accepter tous les appels rentrants les lundis, mardis et jeudis.
- Rejeter tous les appels rentrants les mercredis et les vendredis.

Ces deux politiques ne sont pas en conflit puisqu'elles ne s'exécutent jamais au même moment. Notre méthode reconnait ce cas et n'indique pas l'existence d'un conflit d'action (comme il est mentionné dans le tableau 4.10), puisque les deux actions `accept` et `reject` n'interagissent pas dans cette situation. Les conditions qui déterminent l'exécutabilité des actions sont traduites par des contraintes explicites en Alloy dans la spécification des modèles abstraits des systèmes LESS et APPEL.

Notre méthode formelle est basée d'une part sur l'analyse des actions des politiques, et d'autre part, sur les conditions des politiques.

ACTIONS	connect_to	reject_call	forward_to	fork_to	add_party
connect_to	3,4	1,2,5,6	1,2,6	1,2	1,4
reject_call	1,2,6	4	4,5	4,6	1,2,3,5,6
forward_to	1,2,6	4,5	4	4,6	1,2,3,6
fork_to	1,2,4	3,6	3,6	3	1,2,4
add_party	1,4,5	1,2,3,6	1,2,3,6	1,2,3	4
remove_party	1,2,5,6	1,4	1,4	1,4,6	2,6
add_medium	1,5	1,3	1,3	1,3	
remove_medium	1,5	1,3	1,3	1,3	
remove_default	1,2,5,6	1,3,4	1,3,4	1,3,4,6	2,6
disconnect	1,2,6	1,4	1,4,5	1,4,6	2,3,5,6

	remove_party	add_medium	remove_medium	remove_default	disconnect
connect_to	1,2,6	1	1	1,2,6	1,2,5,6
reject_call	1,3,4,5	1,3,5	1,3,5	1,3,4,5	1,3,4
forward_to	1,3,4	1,3	1,3	1,3,4	1,3,4,5
fork_to	1,6	1	1	1,6	1,6
add_party	2,6			2,6	2,6
remove_party	4			4	4
add_medium		4	2,6		
remove_medium		2,6	4		
remove_default	4			4	4
disconnect	3,4,5	3,5	3,5	3,4,5	2,4

1 : Conflit de simultanéité - état de l'appel
2 : Conflit de simultanéité - état du terminal
3 : Conflit de neutralisation - état de l'appel
4 : Conflit de neutralisation - état du terminal
5 : Conflit de résultat - état de l'appel
6 : Conflit de résultat - état du terminal

TAB. 5.8 – Vérification des conflits du système APPEL

Chapitre 6

Méthode logique pour la détection des interactions

6.1 Introduction

Nous avons vu dans le chapitre 3 que plusieurs efforts pour la détection des interactions de fonctionnalités ont été basés sur l'utilisation des techniques formelles. Notre recherche est dans la même direction, mais utilise une nouvelle méthode, basée sur la sémantique d'Alloy Analyzer.

Alloy permet de modéliser des systèmes complexes. En effet, il permet de simuler les comportements des systèmes et de valider certaines propriétés. Dans ce travail, nous avons utilisé l'outil Alloy Analyzer, qui nous a permis de vérifier les systèmes LESS et APPEL formellement et à un haut niveau d'abstraction. Alloy est distingué par son afficheur, qui permet de visualiser les solutions et les contre-exemples de la vérification.

Alloy Analyzer a déjà été utilisé pour la spécification et la détection des incohérences du systèmes CPP (A Common Profile for Presence) [19].

Dans ce qui suit, nous introduisons les notions de base de l'outil Alloy. Par la suite, nous expliquons son utilisation à l'aide d'exemples découlants de notre étude.

6.2 Alloy - méthode formelle - : Langage et Outil

Alloy [13] est un systéme d'analyse formelle qui est constitué d'un langage et d'un outil. Alloy Analyzer 3.0 est un outil développé par Software Design Group, de MIT et au Computer Science and Artificial Intelligence Laboratory, pour analyser les modèles écrits en langage Alloy. Toutefois, Alloy est un langage de modélisation [13] basé sur la logique du premier ordre (il faut que le modèle soit fini et de taille raisonnable). Alloy est utilisé pour exprimer des contraintes et des comportements structuraux complexes.

Comme Z [4], auquel il s'inspire, Alloy est un langage déclaratif.

Alloy dispose d'un environnement graphique pour visualiser les objets du système et leurs relations puisqu'elles sont décrites par l'utilisateur. Alloy offre plusieurs représentations des résultats de l'exécution des modèles ; les plus utilisées sont les graphes, les arbres binaires et les tables. Ces représentations notifient l'utilisateur de la validité de ses règles.

Les éléments essentiels de Alloy utilisés dans ce travail sont les suivants :

6.2.1 Signature

Une signature, notée *sig*, est une déclaration d'un type de base et d'un ensemble de relations qui représentent les champs. Ces derniers doivent avoir des types et on peut imposer des contraintes sur leurs valeurs. Une signature peut hériter des champs et des contraintes d'une autre signature. Une signature a la structure suivante :

Une signature peut être précédée par le mot clé "abstract", pour signifier que c'est un objet abstrait, ne possédant pas d'instance propre à lui-même. Dans l'exemple présenté dans la figure 6.1, Alloy ne génère pas une instance pour l'objet "OBaction". Donc, l'objet "Rules" hérite à travers la relation "trigger" (événement déclencheur) des propriétés de l'objet "OBtrigger", ainsi que de tous les champs de l'objet "OBtrigger". L'instanciation de l'objet "Rules" contient la structure de l'objet "OBtrigger". La formule "# action = 2" est une contrainte implicite. Elle est toujours vraie et impose au système de produire exactement deux objets "OBaction" tout au long de l'exécution. Sans cette précision, Alloy génère un nombre arbitraire d'objets "OBaction".

```
abstract sig OBtrigger {  }
abstract sig OBswitch {  }
abstract sig OBaction {
  PreCond : set states,   // PreCond est un ensemble d'objets
  CallState : set states,    states
  DeviceState : set states,
  AttribModif : OBmodifier
    }
sig Rules {
  trigger : one OBtrigger, // un seul événement déclencheur
  switch : lone OBswitch,  // une ou plisieurs condition
  action : some OBaction   // l'ensemble action n'est pas vide
  }{
  # action = 2
  }
```

FIG. 6.1 – Structure d'une signature

6.2.2 Fait

Les faits sont des contraintes vraies pour tout le modèle et tout au long de l'exécution. Un fait noté *fact* contient des contraintes explicites sur les relations et les objets et n'a pas besoin d'être invoqué explicitement. Un *fact* ne requière pas d'argument. La figure 6.2 montre un exemple de *fact* qui contient trois contraintes et on impose aux champs de l'objet `accept` d'avoir des valeurs précises. Ces valeurs doivent être compatibles avec le type des champs.

```
fact {
  accept.PreCond = a + b
  accept.CallState = c + d
  accept.DeviceState = e

  reject.PreCond = a
  reject.CallState = c
  reject.DeviceState = f
     }
```

a : incoming call setup pending, b : media devices available
c : call setup finalized, d : a session is setup
e : media devices occupied, f : no change

FIG. 6.2 – Exemple d'un fait

Le symbole "+" dénote l'union d'ensembles en Alloy (voir section 6.2.6).

6.2.3 Prédicat

Un prédicat noté *pred* a la structure suivante :

```
pred Conflict ( a1: OBaction, a2: OBaction )
     {
  !SameAction (a1, a2) and some v: a1.CallState, w: a2.CallState
  | ( v -> w ) in  IncompSet.conflict_incompatibilities
     }
```

FIG. 6.3 – Exemple d'un prédicat

Un prédicat requière des arguments et retourne une réponse booléenne. Si les entrées du prédicat "Conflict" de la figure 6.3 satisfont les contraintes déclarées dans tout le modéle Alloy, alors le prédicat est évalué à vrai, sinon il est jugé faux. Dans le cas où le resultat est faux, l'outil Alloy retourne un contre-exemple.

Le prédicat "Conflit" de la figure 6.3 exprime un comportement conflictuel pour le *conflit d'action* (voir chapitre 4 section 4.4.1). "SameAction" est un prédicat qui vérifie si deux actions ont la même valeur. Si les entrées $a1$ et $a2$ (c.f. *action*1 et *action*2) du prédicat "Conflit" n'ont pas la même valeur et si les post-conditions de $a1$ et $a2$ respectivement appartiennent à l'ensemble des incompatibilités du *conflit d'action*, (voir tableau 4.2) alors il y a un *conflit d'action* entre $a1$ et $a2$.

6.2.4 Assertion

Une assertion est notée *assert* et spécifie une propriété, dont la véracité doit être validée, en considérant les faits et les contraintes implicites du modèle. *Assert* prétend qu'une propriété est vraie, par rapport au comportement du système. L'outil Alloy Analyzer cherche les instances qui satisfont l'assertion. En cas d'existence d'instance, l'outil retourne la réponse "valide". Dans le cas contraire, il affiche un contre-exemple.

```
assert C12 {Conflict(accept, reject)}
check C12 for 11
```

FIG. 6.4 – Exemple d'assertion

L'assertion C12 (voir figure 6.4) suppose que le prédicat "Conflict" est vrai pour les deux objets `accept` et `reject`. Alloy ne trouve pas de contre-exemple (c.-à-d. `accept` et `reject` vérifient le prédicat "Conflict"). Cette assertion est donc valide. Ce qui implique, qu'il y a un conflit entre les actions `accept` et `reject`.

6.2.5 Fonction

Une fonction est une contrainte paramétrable, ayant des arguments et retournant une valeur, le cas échéant un ensemble de valeurs. Au cours de l'exécution du modèle, une fonction notée *fun* est toujours vraie pour un ensemble de solutions possibles. Généralement, une *fun* est utilisée afin de mettre en évidence un ensemble de valeurs, qui vérifient certaines contraintes. Nous proposons un exemple de fonction dans la figure 6.5 pour illustrer l'utilisation des fonctions.

```
fun VerifActions ( r1 : one Rules, x1 : one Rules.action,
                   x2 : one Rules.action ) : Decision
                   {
 if Conflict ( x1, x2 ) then ActionConflict
 else if AttributConflict ( x1, x2 ) then AttConf
 else if ResCompetConflict ( r1.trigger, x1, x2 ) then ResComp
 else if DisablingConflict ( x1, x2 ) then Disabling
 else if EnablingInteraction ( x1, x2 ) then Enabling
 else NoConflict
      }
```

FIG. 6.5 – Exemple de fonction

Dans la figure 6.5, on définit une fonction "VerifActions", ayant en entrée une seule instance des objets nommés $x1, x2$ et $r1$. La fonction produit en sortie une seule valeur de type "Decision". Le corps de la fonction contient des tests de certaines propriétés (des prédicats). Si les arguments en entrée ne vérifient aucun des prédicats alors on aura en sortie la valeur "NoConflict" qui est de type "Decision". Il n'y a aucun moyen avec Alloy de détecter tous les cas conflictuels pour une politique. Même avec la fonction "VerifActions", la vérification s'arrête au premier prédicat satisfait.

6.2.6 Opérateurs logiques

Les opérateurs d'ensemble

Les opérateurs d'ensemble sont consacrés aux atomes (voir figure 6.2). Les opérateurs d'ensemble dans Alloy sont notés comme suit :

Union	+
Intersection	&
Différence	-
Sous-ensemble	in
Égalité	=

Les opérateurs logiques

Les opérateurs logiques dans Alloy sont notés comme suit :

not	!	négation
and	&&	conjonction
or	||	disjonction
implies	=>	implication
else	,	alternative
iff	<=>	double implication

6.3 Conception du modèle Alloy

Cette section présente les signatures de base, les prédicats et le raisonnement adopté dans la conception du modèle Alloy. En outre, nous expliquerons la correspondance entre les éléments du modèle Alloy et les éléments des systèmes téléphoniques décrits dans les chapitres 4 et 5. Les modèles complets des systèmes LESS et APPEL sont fournis et commentés en **Annexe A** et **Annexe B** respectivement.

Une règle de décision est un ensemble de règles, d'une seule décision et d'une cause de la décision prise.

En effet les règles expriment les besoins des utilisateurs. Une décision est le résultat de l'analyse de certaines propriétés. Finalement, la cause indique les actions qui provoquent un conflit, le cas échéant. Une cause est un ensemble d'actions qui pourrait être vide dans le cas où il n'y a pas de conflit. Une règle est constituée d'un événement déclencheur, d'un ensemble de conditions qui pourrait être vide et d'un ensemble d'actions (voir figure 6.1 de la section 6.2.1). Dans cette étude, nous ne considérons que les conflits entre deux actions. Nous imposons au système l'existence d'au moins une action par règle.

Dans le cas du système LESS, notre abstraction prend en considération trois événements déclencheurs, neuf conditions, huit actions et trois attributs pour les actions. La figure 6.6 présente l'énumération de ces quatre ensembles. Tout au long de l'exécution du système, nous aurons une seule instance de chaque élément de ces ensembles.

```
one sig incoming, outgoing, timer extends OBtrigger { }
one sig saddress, spriority, sstring, slanguage, sstatus,
snumberofcalls, spresence, smood, sactivity extends OBswitch { }
one sig accept, transfer, reject, redirect, call, MedUp,
merge, terminate extends OBaction {  }
one sig mlocation, mlookup, mremovelocation extends OBmodifier {  }
```

FIG. 6.6 – Éléments de règles

L'objectif de ce travail est de détecter les conflits qui sont représentés dans la figure 6.7.

```
sig AttConf, Disabling, ResComp, Enabling,
ActionConflict, NoConflict extends Decision { }
```

FIG. 6.7 – Décisions d'analyse

Les objets "AttConf", "Disabling", "ResComp", "Enabling", "ActionConflict", "NoConflict" correspondent respectivement aux Conflit d'attribut, Conflit de neutralisation, Conflit de concurrence sur les ressources, Interactions permises, Conflit d'action et pas de conflit.

Chaque action possède des pré-conditions et des post-conditions qui sont illustrées par des faits (voir figure 6.2 de la section 6.2.2). Les pré et post-conditions doivent être toujours vraies pendant l'exécution du modèle.

Nous rappelons que seules les actions peuvent causer des conflits et cela est dû à l'incohérence de leurs pré et post-conditions. Ces incohérences correspondent aux tableaux des incompatibilités (tableaux 4.2, 4.5, 4.7 du chapitre 4). Ces tableaux sont représentés dans notre modèle comme le montre la figure 6.8.

```
fact name {
 IncompatibilitiesSet.disabling_incompatibilities = c -> h + g -> h
 IncompatibilitiesSet.conflict_incompatibilities = c -> c + g -> k +
  g -> i + g -> g + k -> g + k -> i + i -> g + i -> k

 IncompatibilitiesSet.res_comp_incompatibilities1 = c -> b + d -> b
 IncompatibilitiesSet.res_comp_incompatibilities2 = d -> c + d -> d
 IncompatibilitiesSet.res_comp_incompatibilities3 = d -> d

 }
```

a : incoming call setup pending, b : media devices available, c : call setup finalized
d : a session is setup, e : media devices occupied, f : no change
g : all sessions terminated, h : one or more sessions, i : all sessions alive
j : media transmission changed, k : all sessions merged

FIG. 6.8 – Ensembles d'incompatibilités

Finalement, les prédicats représentent les conflits. Nous citons le conflit d'actions qui est représenté dans la figure 6.3 de la section 6.2.3. Ainsi, on vérifie si deux actions satisfont le corps du prédicat " Conflict " en s'assurant que les pré et post-conditions des actions figurent dans l'ensemble des incompatibilités décrit dans la figure 6.8. Pour cela, nous utiliserons les assertions (voir figure 6.4 de la section 6.4) afin de forcer le système à vérifier toutes les combinaisons possibles des actions.

6.4 L'analyse en Alloy

Alloy Analyzer combine une méthode de description formelle et de vérification de certaines propriétés [13]. Un modèle en Alloy est une abstraction logicielle du système à analyser. Alloy Analyzer nous permet de simuler le comportement d'un système basé sur une sémantique opérationnelle.

6.4.1 Les instances

Un modèle Alloy est une collection de plusieurs instances [13]. Une instance affecte des valeurs aux variables d'un modéle. Avec Alloy, une instance représente un état, une paire d'états ou une trace d'exécution.

Types d'instances dans Alloy

Les modéles de Alloy contiennent deux types d'instances : les instances noyau et les instances des fonctions et des prédicats.

Un modèle sans instances noyau est erroné et dit incohérent. Les instances noyau sont associées aux faits du modèle et aux contraintes implicites dans les signatures. Les instances noyau ainsi que leurs variables saisissent les signatures et leurs champs afin de leur affecter des valeurs, engendrant des faits et déclarations vraies. De plus, les fonctions et les prédicats doivent avoir des instances pour que le modèle soit cohérent, autrement, ils seront inutiles. Les instances noyau doivent être vraies, combinées avec les instances des fonctions et prédicats.
Enfin, les assertions ne sont pas supposées avoir des instances pendant l'exécution du modèle Alloy. Donc, une assertion est utilisée pour exprimer un comportement non considéré dans le restant du model Alloy. L'utilité des assertions est de vérifier si une propriété donnée est valide pour un modèle, sinon l'outil Alloy Analyzer propose un contre-exemple.
Cette technique est utilisée dans ce travail pour la détection des interactions entre politiques des systèmes Less et Appel.

Recherche d'instances

L'outil Alloy Analyzer cherche automatiquement les instances d'un modèle pour un nombre fini d'états "scope" spécifié par l'utilisateur. Donc, si la recherche ne réussit pas à retrouver des instances dans un modèle, cela n'implique pas qu'il n'en existe pas. C'est juste qu'il n'en existe pas pour cette limite d'états.

```
run ConflictRule for 11 but 1 DecisionRules, 1 Rules
```

Fig. 6.9 – Recherche d'instances

La figure 6.9 présente une commande qui exécute le prédicat "ConflictRule" en cherchant les instances qui vérifient ses propriétés, pour onze états, un seul objet "DecisionRules" et un seul objet "Rules".

6.4.2 Logique relationnelle

Alloy est basé sur la logique relationnelle du premier ordre [13]. La philosophie d'Alloy est de décrire un système en termes d'ensembles ayant des relations entre eux. Un atome est l'ensemble le plus petit dans un modèle et vérifie les propriétés suivantes :

- **Indivisible** : n'est pas décomposable ;
- **Inaltérable** : ses propriétés ne changent pas au cours du temps ;
- **Non-Interprété** : ne peut pas intégrer d'autres propriétés.

Par analogie avec la programmation orientée objet, un atome est un objet dans la logique de Alloy. Ainsi, un atome ne peut être qu'en relation avec un ou plusieurs atomes. Les relations comme l'affectation de valeurs aux variables et aux expressions sont des relations entre atomes.

Alloy ne contient pas de notion explicite pour représenter des ensembles et des grandeurs scalaires, mais ceux-ci sont faciles à simuler (voir figure 6.1).

En déclarant une signature, on crée implicitement un atome et les champs définissent les relations de celui-ci avec d'autres atomes (ou signatures). Alloy est flexible dans sa description des systèmes.

En effet, Alloy permet la création de relations unaires, binaires et ternaires. Dans ce travail, nous nous limitons aux relations unaires et binaires vu le temps d'exécution relativement long pour les relations ternaires.

6.5 Validation

Dans ce travail, nous avons considéré huit actions du système Less. Nous avons analysé toutes les combinaisons possibles de couples d'actions (64 combinaisons).
Nous avons testé chaque combinaison d'actions afin de vérifier si elles satisfont ou non

certaines propriétés.

Nous avons proposé cinq types de comportements à vérifier dans le système LESS (voir sections 4.4 du chapitre 4). Donc, notre méthode de détection a englobé tous les cas possibles, a savoir les 320 tests (8 actions * 8 actions * 5 prédicats).

Avec le système APPEL nous avons examiné dix actions. Ainsi, nous avons traité toutes les combinaisons possibles de couples d'actions (100 combinaisons).
Nous avons présenté six types de conflits entre actions à vérifier pour le système APPEL. Par conséquent, nous avons effectué les 600 tests possibles (10 actions * 10 actions * 6 prédicats).

Notre méthode formelle détecte la totalité des cas de conflits d'actions des systèmes APPEL et LESS, par rapport à la caractérisation de conflits d'actions que nous avons proposée et l'abstraction faite des modèles.

6.6 Exécution d'Alloy

Le noyau de l'outil ALLOY exprime les contraintes en termes d'expressions booléennes. Ensuite, il essaye de résoudre ces dernières en invoquant un solutionneur SAT *off-the-shelf SAT*, c.-à-d. un logiciel de satisfaction booléenne déjà disponible. Il est bien connu que les meilleurs algorithmes de satisfaction booléenne disponibles aujourd'hui sont de complexité exponentielle. Cependant, les solutionneurs SAT sont très efficaces et permettent de résoudre beaucoup de problèmes non triviaux.

Les solutionneurs SAT courants peuvent manipuler des milliers de variables et des centaines d'expressions booléennes. Le type et la complexité des expressions influent sur le temps d'exécution [13]. Ainsi, l'utilisateur d'Alloy doit trouver un compromis judicieux entre les détails et l'abstraction. Trop de détails font en sorte que l'exécution échoue à cause d'un dépassement de mémoire ou d'expiration du temps alloué à l'exécution.

Notre méthode peut vérifier les modèles LESS et APPEL comme suit :

– Par la fonction "VerifActions" (voir figure 6.5) qui peut vérifier la totalité du modèle. Cependant, il n'y aura qu'une seule identification de conflit (pour un couple

d'actions, choisie arbitrairement) pour chaque exécution. Alloy ne peut pas chercher tout les cas conflictuels pour un couple d'actions. La vérification s'arrête au premier contre-exemple trouvé ;

– Par la vérification sémantique des assertions (voir figure 6.4). Nous avons considéré toutes les combinaisons possibles des actions du modèle LESS, à savoir les 320 exécutions, alors que, le modèle APPEL en requiert 600 exécutions. Chaque exécution prend à peu près 2.5 minutes pour être vérifiée.

Notre méthode de détection a été exécutée sur un Pentium4 doté de double CPU de 2.80GHz et 1GB de mémoire vive. Nous avons utilisé Alloy Analyzer version 3.

6.7 Conclusion

Ce chapitre présente l'utilisation de l'outil de vérification Alloy pour effectuer l'analyse des conflits entre actions avec les méthodes discutées aux chapitres 4 et 5.

Alloy simule des instances à partir d'un modèle et cherche les contre-exemples relevant de certaines proprietés. L'outil Alloy simule aussi des séquences d'opérations et génère des exemples basés sur le système de spécification.

Il est à noter que le modèle abstrait ne préserve pas nécessairement toutes les propriétés du système concret.

Lorsqu'une propriété est non-valide, l'outil Alloy Analyzer affiche un contre-exemple au niveau abstrait dont la reconstitution au niveau concret peut être laborieuse.

Alloy prend un temps d'exécution considérablement lent. Ce dernier croît exponentiellement avec le nombre d'objets dans le modèle à vérifier. Cependant, il a été possible de compléter l'analyse des conflits pour nos deux systèmes, par rapport aux définitions des conflits adoptées et l'abstraction faite des modèles.

Chapitre 7

Conclusion

7.1 Travail accompli

Dans ce travail, nous avons présenté le probléme des interactions entre politiques de contrôle d'appel en téléphonie sur Internet et ses causes. Nous avons élaboré les approches d'analyse des interactions entre politiques, les plus pertinentes pour notre méthode de détection. Notre méthode formelle se base sur les pré et post-conditions des actions. Nous avons spécifié quatre types de situations conflictuelles au système LESS, à savoir, le *conflit d'action*, le *conflit d'attribut*, le *conflit de concurrence sur les ressources* et le *conflit de neutralisation*. Nous avons également spécifié un type d'*interactions permises*. On a utilisé le langage formel Alloy et son outil pour analyser le système LESS et identifier les comportements conflictuels entre ses actions.

Les résultas de cette recherche ont confirmé les conflits proposés par le concepteur de LESS et ont revelé neuf nouveaux cas conflictuels entre actions (voir le tableau 4.10 de la section 4.5). Les nouveaux conflits sont :

- *Conflit d'action* entre les couples d'actions (`terminate` et `transfer`), (`terminate` et `merge`) et (`terminate` et `media-update`).
- *Conflit de neutralisation* entre les couples d'actions (`transfer` et `merge`), (`transfer` et `media-update`) et (`transfer` et `terminate`).
- *Conflit d'attribut* dans le cas d'exécutions successives des actions `merge`, `terminate` et `call`.

Nous avons identifié les pré et post-conditions des actions du système APPEL (voir section 5.2). Nous avons développé une méthode formelle de détection des conflits basée sur les relations entre les différentes pré et post-conditions des actions. Cette méthode consiste à définir formellement six types de comportements conflictuels et de vérifier si un couple d'actions donné correspond à l'un de ces types.

Nous avons aussi développé la démarche à suivre pour analyser les interactions dans le système APPEL, en utilisant une méthode semblable à celle utilisée avec LESS, (voir section 5.3).

Notre méthode de détection a été appliquée à un modèle abstrait du système APPEL. Par conséquent, la méthode a permis la détection de deux-cent-cinq possibilités de conflits entre les actions du système APPEL (voir section 5.7). Tant dans le cas de LESS, que dans le cas de APPEL, notre analyse est complète, dans le sens où elle considère tous les cas possibles par rapport au niveau d'abstraction choisi.

Notre méthode offre la possibilité de détecter les conflits entre les actions des politiques des services téléphoniques au moment de leur création (voir section 6.6).

Notre méthode de détection facilite l'introduction et la vérification de nouvelles options des systèmes téléphoniques. Souvent, les concepteurs enrichissent leurs systèmes téléphoniques par des nouveaux services qui se traduisent par des nouvelles actions. Pour détecter les nouveaux conflits qui peuvent apparaître, il suffit d'attribuer des pré et post-conditions aux nouvelles actions.

7.2 Travaux futurs

Bien que, nous ayons accompli un certain nombre de réalisations mentionnées dans la section des contributions, d'autres améliorations sont possibles. Cette section présente quelques directives de recherche futures.

7.2.1 Détection des conflits entre politiques sur des terminaux distants

Notre méthode de détection de conflits est appliquée sur des politiques exécutées sur un même terminal. Une façon d'étendre notre travail est de considérer les conflits qui peuvent se produire lors de l'exécution simultanée des politiques par différents utilisateurs sur des terminaux distants (exemple : les politiques administrateurs en présence des politiques utilisateurs). Il semble que l'étude du problème des conflits entre politiques sur différents terminaux peut être envisagé en généralisant notre méthode.

7.2.2 Résolution des conflits détectés

Notre méthode de détection n'est qu'un point de départ pour solutionner ou contourner les conflits. Une deuxième étape serait de suggérer des solutions aux conflits détectés. Le but de cette direction de recherche sera, d'une part, de donner la chance aux usagers d'intervenir par des mesures correctives sur les politiques incohérentes, et d'autre part, d'automatiser la résolution de certains conflits. Pour aider à la résolution des conflits détectés, nous proposons les étapes suivantes :

- Intégration de la méthode de détection présentée dans le chapitre 4, afin d'empêcher les conflits dans le système Less.
- Application de la méthode présentée dans le chapitre 5 sur le modèle complet du système Appel, avec l'élaboration des pré-conditions et des post-conditions pour toutes les actions.
- Intégration de la méthode de détection présentée dans le chapitre 5 au système Appel.

Annexe A

Modèle Alloy du système LESS

```
module models/LESS

/* Structure des règles de décision. */
abstract sig DecisionRules {

/* Une règle de décision est de type règle. */
decisionrule : Rules,

/* La vérification d'une règle de décision engendre une décision. */
testdecision : Decision,

/* La cause de la décision est un ensemble d'actions. */
cause : set OBaction
}

/* Structure des règles */
abstract sig Rules {
/* Une règle est composée d'un trigger, d'un ensemble (qui
peut être vide) de contraintes et d'un ensemble non vide d'actions. */

trigger : one OBtrigger,
switch : lone OBswitch,
action : some OBaction }{

/* Contrainte sur le nombre d'actions. Cette contrainte force le
système à générer exactement deux actions. */
# action = 2
}

sig Decision { }

/* Une décision peut être un Conflit d'attribut, de neutralisation,
 de concurrence sur les ressources, d'action ou pas de conflit. */

sig AttConf, Disabling, ResComp, Enabling,
```

```
ActionConflict, NoConflict extends Decision { }

sig states {  }

/* les pre et post-conditions des actions sont des états de l'appel téléphonique. */
/* a : incoming call setup pending */
/* b : media devices available     */
/* c : call setup finalized        */
/* d: a session is setup           */
/* e: media devices occupied       */
/* f: no change                    */
/* g: all sessions terminated      */
/* h: one or more sessions         */
/* i: all sessions alive           */
/* j: media transmission changed   */
/* k: all sessions merged          */

one sig a, b, c, d, e, f, g, h, i, j, k extends states { }


/* Déclaration des ensembles d'incompatibilités. */
sig IncompatibilitiesSet {
disabling_incompatibilities : states -> states,
conflict_incompatibilities : states -> states,
res_comp_incompatibilities1 : states -> states,
res_comp_incompatibilities2 : states -> states,
res_comp_incompatibilities3 : states -> states
}

sig ChangeState{
var : set states
}


/* Déclaration des ensembles de compatibilités. */
sig CompatibilitiesSet {
Enabling_compatibilities_device1 : states -> states,
Enabling_compatibilities_device2 : states -> states,
Enabling_compatibilities_call1 : states -> states,
Enabling_compatibilities_call2 : states -> states
}

sig NoChangeState{
var_callState_a1 : set states,
var_callState_a2 : set states
}

/* Déclaration des faits entre pre et post-conditions. */
fact name {

 ChangeState.var = d + g
```

```
 NoChangeState.var_callState_a1 = k + i
 NoChangeState.var_callState_a2 = d

 IncompatibilitiesSet.disabling_incompatibilities = c -> h + g -> h
 IncompatibilitiesSet.conflict_incompatibilities = c -> c + g -> k +
  g -> i + g -> g + k -> g + k -> i + i -> g + i -> k

 IncompatibilitiesSet.res_comp_incompatibilities1 = c -> b + d -> b
 IncompatibilitiesSet.res_comp_incompatibilities2 = d -> c + d -> d
 IncompatibilitiesSet.res_comp_incompatibilities3 = d -> d

 CompatibilitiesSet.Enabling_compatibilities_device1 = f -> b
 CompatibilitiesSet.Enabling_compatibilities_device2 = j -> b
 CompatibilitiesSet.Enabling_compatibilities_call1 = c -> b
 CompatibilitiesSet.Enabling_compatibilities_call2 = i ->b
 }

/* Une action possède des pre-conditions des post-conditions de
types état d'appel et état du terminal et un modifier qui indique
l'adresse d'exécution de l'action en question. */

abstract sig OBaction {
PreCond : set states,
CallState : set states,
DeviceState : set states,
AttributeModifier : OBmodifier
}

abstract sig OBtrigger { }

abstract sig OBswitch { }

sig Destination { }

one sig Bob, Alice, Jean extends Destination { }

abstract sig OBmodifier {
Attributes : some Destination
}

one sig incoming, outgoing, timer extends OBtrigger { }

one sig saddress, spriority, sstring, slanguage, sstatus,
snumberofcalls, spresence, smood, sactivity extends OBswitch { }

sig stime extends OBswitch { }

lone sig Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday, Sunday extends stime { }

one sig mlocation, mlookup, mremovelocation extends OBmodifier {  }
```

```
one sig accept, transfer, reject, redirect, call, MedUp,
merge, terminate extends OBaction {  }

/* Déclaration des faits des actions. */
fact  {
  accept.PreCond = a + b
  accept.CallState = c + d
  accept.DeviceState = e

  reject.PreCond = a
  reject.CallState = c
  reject.DeviceState = f

  redirect.PreCond = a
  redirect.CallState = c
  redirect.DeviceState = f

  transfer.PreCond = e + h
  transfer.CallState = g
  transfer.DeviceState = b

  merge.PreCond = h + e
  merge.CallState = k
  merge.DeviceState = e
  MedUp.PreCond = h + e
  MedUp.CallState = i
  MedUp.DeviceState = j

  terminate.PreCond = h + e
  terminate.CallState = g
  terminate.DeviceState = b

  call.PreCond = b
  call.CallState = d
  call.DeviceState = e
  }

/* Prédicat du conflit d'action */
pred Conflict ( a1 : OBaction, a2 : OBaction ) {
!SameAction ( a1, a2 ) and some v : a1.CallState,
w : a2.CallState | ( v -> w ) in
IncompatibilitiesSet.conflict_incompatibilities
}
pred SameAction( a1 : OBaction, a2 : OBaction ) {
a1 = a2 and a2 in a1
}
pred DifferentDestinations(a1 : OBaction, a2 : OBaction) {
a1.AttributeModifier.Attributes != a2.AttributeModifier.Attributes
and (a1.AttributeModifier.Attributes not in
a2.AttributeModifier.Attributes)
```

```
}

/* Prédicat du conflit d'attribue. */
pred AttributeConflict (a1 : OBaction, a2 : OBaction) {
SameAction ( a1, a2 ) and DifferentDestinations ( a1, a2 )
  }

/* Prédicat du conflit de concurrence sur les ressources. */
pred ResourceCompetitionConflict ( a1 : OBaction, a2 : OBaction ) {
all p1 : a1.CallState, q1 : a2.PreCond | ( ( p1 -> q1 ) in
IncompatibilitiesSet.res_comp_incompatibilities1 )
or ( all p2 : a1.CallState, q2 : a2.CallState | ( ( p2 -> q2 ) in
IncompatibilitiesSet.res_comp_incompatibilities2 ) )
or ( all p3 : a1.CallState, q3 : a2.CallState | ( ( p3 -> q3 ) in
IncompatibilitiesSet.res_comp_incompatibilities3 ) )
or ( some p4 : a1.CallState, q4 : a2.CallState | p4 in
NoChangeState.var_callState_a1 and q4 in NoChangeState.var_callState_a2 )
}

/* Prédicat du conflit de neutralisation. */
pred DisablingConflict ( a1 : OBaction, a2 : OBaction ) {
some v : a1.CallState, w : a2.PreCond | ( ( v -> w ) in
IncompatibilitiesSet.disabling_incompatibilities ) and
( a1.DeviceState !in a2.PreCond )
}
/* Prédicat des interactions permises. */
pred EnablingInteraction ( a1 : OBaction, a2 : OBaction ) {
( a1.DeviceState in a2.PreCond ) and
( some x : a1.CallState | x in ChangeState.var )
or ( all z1 : a1.DeviceState, t1 : some a2.PreCond,
s1 : a1.CallState | ( ( ( s1 -> t1 ) in
CompatibilitiesSet.Enabling_compatibilities_call1 ) and
(( z1 -> t1 ) in CompatibilitiesSet.Enabling_compatibilities_device1) )  )
or ( some s2 : a1.CallState, z2 : a1.DeviceState,
t2 : a2.PreCond | ( ( ( s2 -> t2 ) in
CompatibilitiesSet.Enabling_compatibilities_call2 ) and (( z2 -> t2 ) in
CompatibilitiesSet.Enabling_compatibilities_device2) )   )
}

/* Exécution des prédicats définis précédemment pour
onze objets (les pre et post-conditions). */

run Conflict for 16

run SameAction for 16

run DifferentDestinations for 16

run AttributeConflict for 16

run ResourceCompetitionConflict for 16
```

```
run DisablingConflict for 16

run EnablingInteraction for 16

//-------------------------------------------------------------//
// -------- function de vérification en temps réel -------- //
//-------------------------------------------------------------//
/*
fun VerifActions ( r1 : one Rules, x1 : one Rules.action ,
x2 : one Rules.action ) : some Decision {

if Conflict ( x1, x2 ) then ActionConflict
else if AttributeConflict ( x1, x2 ) then AttConf
else if ResourceCompetitionConflict ( x1, x2 ) then ResComp
else if DisablingConflict ( x1, x2 ) then Disabling
else if EnablingInteraction ( x1, x2 ) then Enabling
else NoConflict
}
/* Prédicat qui analyse l'existence des conflits pour une règle. */
pred ConflictRule (
r2 : one DecisionRules, y1 : one r2.decisionrule.action,
y2 : one ( r2.decisionrule.action - y1 )) {
 r2.testdecision = VerifActions ( r2.decisionrule, y1, y2 )
 r2.cause = y1 + y2
}
Exécution du prédicats ConflictRule pour
onze objets (les pre et post-conditions).

run ConflictRule for 11 but 1 DecisionRules, 1 Rules
*/

//-------------------------------------------------------------//
// ------------- Vérification par des assertions ------------- //
//-------------------------------------------------------------//


// -------------------- Conflict d'action -------------------- //

/*
assert C11 { Conflict ( accept, accept )  }
assert C12 { Conflict ( accept, reject )  }
assert C13 { Conflict ( accept, redirect ) }
assert C14 { Conflict ( accept, transfer )  }
assert C15 { Conflict ( accept, merge )  }
assert C16 { Conflict ( accept, MedUp )  }
assert C17 { Conflict ( accept, terminate ) }
assert C18 { Conflict ( accept, call )   }
check C11 for 16
check C12 for 16
check C13 for 16
```

```
check C14 for 16
check C15 for 16
check C16 for 16
check C17 for 16
check C18 for 16

...
*/
```

Annexe B

Modèle Alloy du système APPEL

```
module models/APPEL

abstract sig DecisionRules {
 decisionrule : Rules,
 testdecision : Decision,
 cause : set OBaction
    }

abstract sig Rules {
 trigger : one OBtrigger,
 condition : lone OBcondition,
 action : some OBaction
  }{
  # action = 2
  }

sig Decision { }

sig ConcurrencyConflictCS, ConcurrencyConflictMS, DisablingConflictCS,
DisablingConflictMS, ResultsConflictCS, ResultsConflictMS,
NoConflict extends Decision { }

sig states {  }

/* Ensemble des pre et post-conditions des actions. */
one sig NotCallExists, CallSetup, CallExists, PartyAddedToCall,
CallForwarded, CallForked, DefaultAvailable, DefaultReserved,
MediumAvailable, MediumReserved extends states { }

sig IncompSet {
 ConcConflict_incomp_ConnState : states -> states,
 ConcConflict_incomp_MedState  : states -> states,

 DisConflict_incomp_ConnState : states -> states,
 DisConflict_incomp_MedState : states -> states,
```

```
 ResConflict_incomp_ConnState : states -> states,
 ResConflict_incomp_MedState : states -> states
 }

/* Déclaration des ensembles d'incompatibilités entre les pre et post-conditions
des actions. */
fact AC {
IncompSet.ConcConflict_incomp_ConnState = CallExists -> NotCallExists +
CallExists -> CallSetup + NotCallExists -> CallExists + NotCallExists -> CallSetup +
CallSetup -> CallExists + CallSetup -> NotCallExists

IncompSet.ConcConflict_incomp_MedState  = DefaultAvailable -> DefaultReserved +
DefaultReserved -> DefaultAvailable + MediumAvailable -> MediumReserved +
MediumReserved -> MediumAvailable

IncompSet.DisConflict_incomp_ConnState = NotCallExists -> CallExists +
CallSetup -> NotCallExists + CallExists -> CallSetup

IncompSet.DisConflict_incomp_MedState = DefaultReserved -> DefaultAvailable +
DefaultAvailable -> DefaultReserved + MediumAvailable -> MediumReserved +
MediumReserved -> MediumAvailable

IncompSet.ResConflict_incomp_ConnState = CallExists -> CallSetup +
NotCallExists -> CallExists + CallSetup -> NotCallExists +
NotCallExists -> CallForwarded + CallForwarded -> NotCallExists

IncompSet.ResConflict_incomp_MedState = DefaultAvailable -> DefaultReserved +
DefaultReserved -> DefaultAvailable + MediumAvailable -> MediumReserved +
MediumReserved -> MediumAvailable
}

abstract sig OBaction {
  PreConnectionState : set states,
  PreMediaState : set states,
  PostConnectionState : set states,
  PostMediaState : set states
  }

abstract sig OBcondition { }

abstract sig OBtrigger { }

one sig connIncom, connOut extends OBtrigger { }

one sig date, time, callee, caller, networktype, bandwidth,
device extends OBcondition { }

sig day extends OBcondition { }

lone sig Monday, Tuesday, Wednesday, Thursday, Friday,
```

```
Saturday, Sunday extends day { }

one sig connect_to, reject_call, forward_to, fork_to, add_party, remove_party,
add_medium, remove_medium, remove_default, disconnect extends OBaction { }

/* Déclaration des faits des actions. */
fact  {
  connect_to.PreConnectionState = NotCallExists
  connect_to.PreMediaState = DefaultAvailable
  connect_to.PostConnectionState = CallSetup
  connect_to.PostMediaState = DefaultReserved

  reject_call.PreConnectionState = CallSetup
  reject_call.PreMediaState = DefaultReserved
  reject_call.PostConnectionState = NotCallExists
  reject_call.PostMediaState = DefaultAvailable

  forward_to.PreConnectionState = CallSetup
  forward_to.PreMediaState = DefaultReserved
  forward_to.PostConnectionState = CallForwarded
  forward_to.PostMediaState = DefaultAvailable

  fork_to.PreConnectionState = CallSetup
  fork_to.PreMediaState = DefaultReserved
  fork_to.PostConnectionState = CallForked
  fork_to.PostMediaState = DefaultReserved

  add_party.PreConnectionState = CallExists
  add_party.PreMediaState = DefaultAvailable
  add_party.PostConnectionState = PartyAddedToCall + CallExists
  add_party.PostMediaState = DefaultReserved

  remove_party.PreConnectionState = CallExists + PartyAddedToCall
  remove_party.PreMediaState = DefaultReserved
  remove_party.PostConnectionState = CallExists
  remove_party.PostMediaState = DefaultAvailable

  add_medium.PreConnectionState = CallExists
  add_medium.PreMediaState = MediumAvailable
  add_medium.PostConnectionState = CallExists
  add_medium.PostMediaState = MediumReserved

  remove_medium.PreConnectionState = CallExists
  remove_medium.PreMediaState = MediumReserved
  remove_medium.PostConnectionState = CallExists
  remove_medium.PostMediaState = MediumAvailable

  remove_default.PreConnectionState = CallExists
  remove_default.PreMediaState = DefaultReserved
  remove_default.PostConnectionState = CallExists
  remove_default.PostMediaState = DefaultAvailable
```

```
  disconnect.PreConnectionState = CallExists
  disconnect.PreMediaState = DefaultReserved
  disconnect.PostConnectionState = NotCallExists
  disconnect.PostMediaState = DefaultAvailable
  }

/* Prédicat Conflit de simultanéité état de l'appel */
pred Conc_Confl_ConnState ( a1 : OBaction, a2 : OBaction ) {
some v : a1.PreConnectionState , w : a2.PreConnectionState | (v -> w) in
IncompSet.ConcConflict_incomp_ConnState
 }

/* Prédicat Conflit de simultanéité état du terminal*/
pred Conc_Confl_MedState ( a1 : OBaction, a2 : OBaction ) {
some v : a1.PreMediaState , w : a2.PreMediaState | (v -> w) in
IncompSet.ConcConflict_incomp_MedState
 }

/* Prédicat Conflit de neutralisation état de l'appel */
pred Dis_Confl_ConnState ( a1 : OBaction, a2 : OBaction ) {
some v : a1.PostConnectionState , w : a2.PreConnectionState | (v -> w) in
IncompSet.DisConflict_incomp_ConnState
 }

/* Prédicat Conflit de neutralisation état du terminal */
pred Dis_Confl_MedState ( a1 : OBaction, a2 : OBaction ) {
some v : a1.PostMediaState , w : a2.PreMediaState | (v -> w) in
IncompSet.DisConflict_incomp_MedState
 }

/* Prédicat Conflit de résultats état de l'appel */
pred Res_Confl_ConnState ( a1 : OBaction, a2 : OBaction ) {
some v : a1.PostConnectionState , w : a2.PostConnectionState | (v -> w) in
IncompSet.ResConflict_incomp_ConnState
 }

/* Prédicat Conflit de résultats état du terminal */
pred Res_Confl_MedState ( a1 : OBaction, a2 : OBaction ) {
some v : a1.PostMediaState , w : a2.PostMediaState | (v -> w) in
IncompSet.ResConflict_incomp_MedState
 }

//-------------------------------------------------------//
// -------- function de vérification en temps réel -------- //
//-------------------------------------------------------//

/*
fun VerifActions ( r1 : one Rules, x1 : one Rules.action , x2 : one Rules.action ) :
some Decision {
```

```
if Conc_Confl_ConnState ( x1, x2 ) then ConcurrencyConflictCS
else if Conc_Confl_MedState ( x1, x2 ) then ConcurrencyConflictMS
else if Dis_Confl_ConnState ( x1, x2 ) then DisablingConflictCS
else if Dis_Confl_MedState ( x1, x2 ) then DisablingConflictMS
else if Res_Confl_ConnState ( x1, x2 ) then ResultsConflictCS
else if Res_Confl_MedState ( x1, x2 ) then ResultsConflictMS
else NoConflict
 }
pred ConflictRule ( r2 : one DecisionRules,
y1 : one r2.decisionrule.action, y2 : one ( r2.decisionrule.action - y1 ))
  {
r2.testdecision = VerifActions ( r2.decisionrule, y1, y2 )
r2.cause = y1 + y2
  }

run ConflictRule for 10 but 1 DecisionRules, 1 Rules
*/

//-----------------------------------------------------------//
// ----------- Vérification par des assertions ------------ //
//-----------------------------------------------------------//


// --------- Conflit de simultanéité état de l'appel --------- //
/*
assert C11 { Conc_Confl_ConnState ( connect_to, connect_to )  }
assert C12 { Conc_Confl_ConnState ( connect_to, reject_call )  }
assert C13 { Conc_Confl_ConnState ( connect_to, forward_to ) }
assert C14 { Conc_Confl_ConnState ( connect_to, fork_to )  }
assert C15 { Conc_Confl_ConnState ( connect_to, add_party )  }
assert C16 { Conc_Confl_ConnState ( connect_to,  remove_party)  }
assert C17 { Conc_Confl_ConnState ( connect_to, add_medium ) }
assert C18 { Conc_Confl_ConnState ( connect_to, remove_medium )   }
assert C19 { Conc_Confl_ConnState ( connect_to, remove_default )   }
assert C1A { Conc_Confl_ConnState ( connect_to, disconnect  )   }

check C11 for 17
check C12 for 17
check C13 for 17
check C14 for 17
check C15 for 17
check C16 for 17
check C17 for 17
check C18 for 17
check C19 for 17
check C1A for 17
*/
```

Bibliographie

[1] Aiguier, M., Berkani, K., and Gall, P. L. Feature specification and static analysis for interaction resolution. *Lecture Notes in Computer Science 4085*, 25 (2006), 364–379.

[2] Amyot, D., Gray, T., Liscano, R., Logrippo, L., and Sincennes, J. Interactive conflict detection and resolution for personalized features. *Journal of communication and networks 7*, 3 (2005), 353–366.

[3] Boukhors, A. *XML : la synthèse : intégrez XML dans vos architectures.* Paris : Dunod, 2002.

[4] Bowen, J. P., and Hinchey, M. G. *The Z Formal Specification Notation.* Springer, 1995.

[5] Calder, M., Kolberg, M., Magill, E. H., and Reiff-Marganiec, S. Feature interaction : a critical review and considered forecast. *Comput. Networks 41*, 1 (2003), 115–141.

[6] Cameron, J., Griffeth, N., Lin, Y., Nilson, M., Schnure, W., and Velthuijsen, H. A feature-interaction benchmark for in and beyond. *Communications Magazine, IEEE 31*, 3 (1993), 64–69.

[7] Cameron, J., and Velthuijsen, H. Efficient and pratical techniques can be devised to identify and manage adverse feature interactions. *IEEE, Computer society 31* (1993), 18– 23.

[8] Campbell, G. A., and Turner, K. J. Policy conflict filtering for call control. In *9th International Conference on Feature Interaction in software and communication systems* (Grenoble, France, Septembre 2007), ICFI, pp. 93–108.

[9] Felty, A. P., and Namjoshi, K. S. Feature specification and automated conflict detection. *ACM Trans. Softw. Eng. Methodol. 12*, 1 (2003), 3–27.

[10] Gorse, N., Logrippo, L., and Sincennes, J. Formal detection of feature interactions with logic programming and lotos. *Software and Systems Modeling 5*, 2 (June 2006), 121–134.

[11] Hardin, R. H., Har'El, Z., and Kurshan, R. P. Cospan (coordination specification analizer). In *8th International Conference, Computer Aided Verification, 96* (New Brunswick, NJ, USA, August 1996), Lecture Notes in Computer Science, pp. 423–427.

[12] Hay, J. D., and Atlee, J. M. Composing features and resolving interactions. In *SIGSOFT '00/FSE-8 : Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2000), ACM Press, pp. 110–119.

[13] Jackson, D. *Software Abstractions : Logic, Language and Analysis.* The MIT Press, 2006.

[14] Jackson, M., and Zave, P. Distributed feature composition : A virtual architecture for telecommunications services. *IEEE Trans. Softw. Eng. 24*, 10 (1998), 831–847.

[15] Johnston, A. *SIP : Understanding the Session Initiation Protocol.* Artech House, Inc., Boston, London, 2000.

[16] Johnston, A. *SIP : Understanding the Session Initiation Protocol, Second Edition.* Artech House, Inc., Norwood, MA, USA, 2003.

[17] Jonathan Lennox, W. X., and Schulzrinne, H. Call processing language (cpl) : A language for user control of internet telephony services. `http://www.ietf.org/rfc/rfc3880.txt`, October 2004.

[18] Kolberg, M., Magill, E., and Wilson, M. Compatibility issues between services supporting networked appliances. *Communications Magazine 41*, 11 (2003), 136–147.

[19] Lau, E. Case study in alloy modeling : A common profile for presence. `http://sdg.csail.mit.edu/6.894/assignments/instant-messaging.pdf`, 2003.

[20] Layouni Ahmed Fadhel, L. L., and Turner, K. J. Conflict detection in call control using first-order logic model checking. In *9th International Conference on Feature Interaction in software and communication systems* (Grenoble, France, Septembre 2007), ICFI, pp. 77–92.

[21] Nakamura, M., Leelaprute, P., ichi Matsumoto, K., and Kikuno, T. On detecting feature interactions in the programmable service environment of internet telephony. *Comput. Networks 45*, 5 (Mars 2004), 605–624.

[22] Noy, N. F., and McGuinness, D. L. A guide to creating your first ontology. `http://www.ksl.stanford.edu/KSL_Abstracts/KSL-01-05.html`, Septembre 2001.

[23] Peterson, J. `http://tools.ietf.org/html/rfc3859`, August 2004.

[24] Reiff-Marganiec, S., Turner, K. J., and Blair, L. Appel : The accent project policy environment/language. `http://www.cs.stir.ac.uk/~kjt/techreps/pdf/TR161.pdf`, December 2005.

[25] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and Schooler, E. Sip : Session initiation protocol, rfc 3261. `http://www.ietf.org/rfc/rfc3261.txt`, June 2002.

[26] Stephan Reiff-Marganiec, K. J. T., and Blair, L. Appel : The accent project policy environment/language. `http://www.cs.stir.ac.uk/research/publications/techreps/pdf/TR161.pdf`, December 2005.

[27] Tsuchiya, T., Nakamura, M., and Kikuno, T. Detecting feature interactions in telecommunication services with a sat solver. *prdc 00* (2002), 131.

[28] Turner, K. J., and Blair, L. Policies and conflicts in call control. *Comput. Networks 51*, 2 (2007), 496–514.

[29] Turner, K. J., Reiff-Marganiec, S., Blair, L., Pang, J., Gray, T., Perry, P., and Ireland, J. Policy support for call control. *Computer Standards and Interfaces 6*, 28 (June 2006), 635–649.

[30] UDDI. Universal description descovery and integration. `http://www.uddi.org/`, 2007.

[31] (W3C), T. W. W. W. C. Web ontology language (owl). `http://www.w3.org/TR/owl-ref/`, February 2004.

[32] (W3C), T. W. W. W. C. Extensible markup language (xml). `http://www.w3.org/XML/`, 2007.

[33] (W3C), T. W. W. W. C. Simple object access protocol (soap). `http://www.w3.org/TR/soap`, 2007.

[34] (W3C), T. W. W. W. C. Web services description language (wsdl). http://www.w3.org/TR/wsdl, 2007.

[35] Wikipédia. Service web. http://wikipedia.org/, 2007.

[36] Wu, X., and Schulzrinne, H. Columbia university telecommunication service editor (cute). http://www.cs.columbia.edu/IRT/sipc/doc/html/, 2007.

[37] Wu, X., and Schulzrinne, H. Handling feature interactions in the language for end system services. *Comput. Networks 51*, 2 (2007), 515–535.

[38] Wu Xiaotaow, J. L., and Schulzrinne, H. Cpl extensions for presence. http://www1.cs.columbia.edu/~xiaotaow/rer/Research/Paper/draft-wu-cpl-presence-00.txt, April 2001.

[39] Xu, Y. Detecting feature interactions and feature inconsistencies in cpl. http://cserg0.site.uottawa.ca/ftp/pub/Lotos/Theses/, September 2003.

[40] Xu, Y., Logrippo, L., and Sincennes, J. Detecting feature interactions in cpl. *J. Netw. Comput. Appl. 30*, 2 (2007), 775–799.

Index