Modeling and methodologies for the test of IMS services

Felipe Lalanne Rojas Lalanne

TÉLÉCOM SUDPARIS

ÉCOLE DOCTORALE S&I

EN CO-ACCRÉDITATION AVEC L'UNIVERSITÉ D'ÉVRY-VAL D'ESSONNE

T H È S E

pour obtenir le titre de

Docteur

de Télécom SudParis

Spécialité : INFORMATIQUE

Présentée et soutenue par

Felipe LALANNE

Modélisation et Méthodologie pour le Test de Services IMS

Thèse dirigée par Ana CAVALLI

soutenue le 3 février 2012

Jury :

Rapporteurs:	Fatiha ZAÏDI	-	Université Paris Sud XI
	Mercedes MERAYO	-	Universidad Complutense de Madrid
Directeur:	Ana CAVALLI	-	Télécom SudParis
Examinateurs:	Stéphane MAAG	-	Télécom SudParis
	Jean-Luc RICHIER	-	Laboratoire d'Informatique de Grenoble
	Michel DIAZ	-	CNRS LAAS

Thèse N°2012TELE0003

Acknowledgements

First of all I would like to thank my thesis director, Ana Cavalli, for granting me the opportunity to work with her and the team and all her support with the finalisation of my studies. I also want to thank Stéphane Maag, my supervisor, for all his support and his input during this period, for listening to my ideas and helping me in discussing them. A big thank you also to the reviewers of my thesis Fatiha Zaïdi and Mercedes Merayo, for their comments and kind remarks, as well as the members of the jury, Jean-Luc Richier and Michel Diaz, for taking the time of their schedules for reviewing my work.

On a more personal note, I would like to thank my wife, Victoria, for accompanying me in this adventure, with all the risks it implicated (the least of which was learning a new language) and for her support during all this time, particularly in the last months of my work, in all those late nights of writing.

I'd also like to thank my family for all their support at the distance and the confidence they have always have put in me.

I cannot fail to mention the members of the LOR team, present and those that are already looking for new adventures: Iksoon, Bakr, Fayçal, Gerardo, José Pablo, Mazen, Anderson, Pramila and Alessandra.

Finally, I would also like to thank the members of the chilean team of NIC Labs, particularly José Miguel Piquer and Tomás Barros, for their advice prior to the pursuit of my Ph.D. and for acting as a liaison between me and the LOR team.

Résumé

Contexte Général

Aujourd'hui, quand la communication est essentielle et qu'un nombre très important de services est disponible en ligne, les réseaux informatiques continuent à croître et de nouveaux protocoles de communication sont constamment développés. Dans ce domaine, les normes de communication sont essentielles pour permettre l'interfonctionnement des systèmes. Tandis que les techniques de *vérification formelle* sont utilisées pour établir si les normes sont correctes [Woodcock 2009], la conformité de leurs implémentations aux exigences de la norme est habituellement évaluée par des techniques de *test*.

Le test de conformité est le processus qui contrôle qu'un système possède un ensemble des propriétés souhaitées et se comporte conformément à certaines exigences prédéfinies. On connait son importance et l'impact pour le déploiement et l'utilisation future des logiciels et des systèmes. Ceci est notamment observée dans les nombreux travaux dans le domaine du test; travaux fournis par la communauté recherche [Hierons 2009], mais aussi par l'industrie [ETSI/ES 201 873-1 2007] et les instituts de standardisation [ETSI/ETR 022 1993, ISO/IEC 9646 1994].

Une partie importante des approches de test sont basées sur des *méthodes formelles*. Ces méthodes s'appuient sur des *spécifications formelles*, des modèles mathématiques construits en utilisant les exigences informelles du système au cours de la phase de spécification. La spécification formelle est utilisée pour soutenir le processus de test, par exemple, avec des techniques automatisées de génération des cas de test et des métriques de couverture, il est possible par des expériences de déterminer de manière fiable si une implémentation d'un protocole satisfait ses exigences. Dans un contexte de test de conformité, des méthodes *boîte-noire* (black-box en anglais) son habituellement utilisées, où la structure interne de l'implémentation est inconnue et toute évaluation se fait par l'observation des entrées, sorties et l'environnement de *l'implémentation sous test* (IUT pour Implementation Under Test). Deux mécanismes principaux sont habituellement distingués à cet effet en fonction de leur niveau d'interaction avec l'IUT: le *test passif* et le *test actif*.

Le *test actif* est basé sur l'exécution des séquences de test spécifiques contre une implémentation sous test. Des séquences de test sont générées à partir d'une spécification formelle du protocole sur la base de multiples critères de satisfaction de couverture et de conformité. Les tests peuvent être générés automatiquement ou semi-automatiquement en fonction de différentes hypothèses quant à l'implémentation et des objectifs de test différents. L'exécution des tests est effectuée par le biais des *points de contrôle et d'observation* (PCOs pour Points of Control and Observation), c.-à-d. des interfaces d'exécution. Ces PCOs sont définis dans le cadre d'une ar-

chitecture test, une distribution des testeurs particulière qui leur permet d'interagir avec la plate-forme ainsi que de communiquer entre eux si nécessaire.

Le *test passif* est basé sur l'observation d'événements d'entrée et de sortie d'une implémentation sous test en cours d'exécution. Le terme "passif" implique que les tests ne perturbe pas le fonctionnement naturel du système étant donné que l'implémentation n'est pas stimulée. La suite d'observations d'événements s'appelle une *trace*. Afin de vérifier la conformité de l'IUT, cette trace sera comparée à un ensemble de comportements attendus, définis soit par un modèle formel (si disponible) ou par une ou plusieurs propriétés fonctionnelles (de conformité). Le test passif fournit une alternative au test actif, lorsque les exigences de ce dernier (définition d'architectures de test et accès aux interfaces du système pour contrôler les entrées) sont inaccessibles ou indésirables.

Dans le test passif, des méthodologies de *test à base d'invariants* peuvent être distinguées. Là, un ensemble de propriétés (critiques) de conformité est défini, soit automatiquement à partir d'une spécification formelle, ou manuellement à partir des exigences du système. Ces propriétés spécifient des séquences d'événements qui doivent être observées dans la trace pour établir la conformité de l'implémentation. Le principe est basé sur la relation d'implémentation de *pré-ordre dans la trace* (trace preorder en anglais), c'est-à-dire, si une trace est observée depuis l'IUT qui ne peut pas être produite par la spécification, alors ceci est indicatif d'une faute dans l'implémentation. Dans le test à base d'invariants, le comportement qui doit être observé est défini par les propriétés de conformité (les invariants).

Les méthodologies de test à base d'invariants ont certains avantages par rapport aux techniques de test passif basées sur des spécifications. Les techniques de test passif compare le comportement observé (dans la trace) avec le comportement attendu de la spécification, pour déterminer si le premier peut être produit par ce dernier. Cela exige la vérification de chaque état de la spécification au pire des cas. Les méthodologies liées aux invariants fournissent une méthode pour la détermination rapide de la conformité de propriétés critiques. Ces techniques peuvent aussi être utiles pour le test lorsque la spécification n'est pas disponible, ce qui est souvent le cas pour de grands systèmes. Finalement, des techniques basées sur des invariants fournissent aussi des perspectives intéressantes pour la surveillance des propriétés de conformité lors de l'exécution de l'IUT. Dans ce dernier contexte, un domaine connexe est devenue très populaire cette dernière décennie au sein de la communauté vérification, appelé *runtime verification* [Leucker 2009].

Le runtime verification est une discipline, dérivée du *model checking*, qui s'occupe de l'étude des techniques de vérification qui permettent de vérifier si une exécution d'un système répond à une propriété de correction particulière. Contrairement au test passif, les traces d'exécution en runtime verification ne sont pas limitées qu'aux événements entrée/sortie de l'IUT, et sont généralement décrits comme une séquence d'états du système. Comme avec le model checking, le runtime verification se concentre majoritairement aux aspects techniques de l'évaluation des propriétés et de

la génération des moniteurs. Néanmoins, un bon nombre des techniques et des concepts du runtime verification peut également être utilisé pour le test passif.

Contributions

L'Internet Multimedia Subsystem (IMS) est un système standardisé fournissant des services IP multimédia aux utilisateurs mobiles. Il offre une architecture centralisée pour les opérateurs de télécommunication pour intégrer et fournir l'accès aux multiples services 3G, comme les services de voix et communication multimédias, tout en fournissant des fonctionnalités de contrôle de qualité de service (QoS pour quality of service) et de tarification pour les opérateurs. Les services IMS s'appuient fortement sur les normes de l'IETF, en particulier le Session Initiation Protocol (SIP), utilisé pour le contrôle de session et la communication.

Différemment des autres protocoles, la spécification SIP ne décrit pas un service, mais un ensemble de primitives extensibles pour l'établissement des sessions, la configuration et l'interruption des services, avec plus de 200 standards[1]. Les services IMS intègrent généralement de nombreuses extensions, offrant ainsi des défis intéressants en matière du test de conformité.

Le test de conformité s'occupe généralement d'établir la concordance de l'implémentation d'un protocole par rapport à son standard, néanmoins certaines techniques de test de conformité peuvent aussi être applicables pour tester des implémentations de services. Dans le cas des services IMS, des fonctionnalités de protocoles multiples ainsi que de services multiples sont intégrés et leur intégration est décrite dans des documents standards [Open Mobile Alliance 2006, Open Mobile Alliance 2010]. Le test de conformité dans le cas des services IMS doit alors tenir compte de deux aspects distincts: *1)* conformité de chaque protocole et extension mise en œuvre par le service à sa spécification particulière, *2)* et la conformité de l'intégration des extensions aux exigences du service.

En outre, le manque d'implémentations ouvertes pour les services IMS et la difficulté habituelle de l'accès aux interfaces de service de l'opérateur limitent les possibilités pour leur test actif. Dans ce travail on fournit des améliorations aux techniques de test passif pour le test des services IMS et SIP.

Les techniques traditionnelles de test passif dérivent de techniques de tests basés sur des modèles, comme celles basées sur des machines à états finis (FSM pour Finite State Machine), machines à états finies étendues (EFSM pour Extended FSM) et systèmes de transitions étiquetées (LTS pour Labelled Transition System). Ces modèles présument souvent une relation causale entre les parties de contrôle des entrées et sorties dans les transitions du modèle, en raison de leur utilisation pour des systèmes réactifs, ce qui signifie que les traces (observations) à partir du système

[1] Pour obtenir une liste à jour le lecteur peut consulter `http://www.packetizer.com/ipmc/sip/standards.html`

prennent la forme d'une séquence de couples d'entrée / sortie. Cela permet aux techniques de test, et en particulier aux techniques de test basées sur invariants, de faire usage de la causalité comme pour définir des propriétés, des séquences d'entrées / sorties, qui doivent être vues sur la trace pour établir la conformité de l'implémentation. Pour les traces de nombreux protocoles, telle causalité n'est pas toujours applicable, puisque plusieurs sorties peuvent être attendues pour une entrée et vice-versa. De plus, dans la collecte de traces réelles, en particulier dans le cas des services centralisés, la trace peut contenir des interactions avec de clients multiples, ce qui rend encore plus difficile l'établissement de causalité sur la base de parties de contrôle.

Dans ce type de traces la causalité entre les événements dans une trace peut souvent n'être établie que par les parties de données de messages. Cependant, comme les approches traditionelles de test (et vérification) dérivent de travaux avec des modèles à états finis (ou à transition étiquetée), elles sont généralement de nature propositionnelle, ce qui signifie qu'elles prennent d'abord en compte les parties de contrôle, avec les parties de données comme une extension de parties de contrôle, généralement sous la forme des paramètres ajoutés aux parties de contrôle. Cela signifie une expressivité réduite des formules pour établir des relations sur la base des données, ou l'expressivité au dépens de la brièveté des formules. Cela devient encore plus critique lorsque des contraintes autres que l'égalité des paramètres de données sont nécessaires. Même si certains travaux avec des approches à base d'invariants ont été proposées pour traiter les données sous la forme de contraintes [Ladani 2005], ils exigent l'usage d'une spécification afin de déterminer les contraintes, ce qui limite l'aspect practique de l'approche, car une spécification n'est pas toujours disponible, en particulier pour les grands systèmes.

Un dernier point doit être pris en compte lorsque la causalité entre événements est supprimé. Lors du test avec des propriétés sur des traces finies, pour une propriété telle que "si l'événement x se produit alors un événement y doit être observé sur la trace", il peut se produire que l'événement x est observé, mais l'événement y ne l'est pas. Avec quelques hypothèses au sujet d'exécution, il peut ne pas être possible de distinguer les cas: "l'événement y n'a jamais été produit par l'implémentation" et "la collecte de la trace a fini avant que y a pu être observé". Cette question avait déjà été notée pour des propriétés nommées *backward* [Bayse 2005] ("si x est observé, alors y doit avoir été observé avant"), et une solution basée sur l'utilisation de la technique de *homing phase* du test passif a été proposée pour détecter si l'état initial de la spécification figurait dans la trace. Toutefois, si une spécification n'est pas immédiatement disponible, cette solution n'est pas réalisable, et aucune solution équivalente n'existe pour les propriétés nommées *forward*. Des questions similaires ont été identifiées dans la littérature du runtime verification [Bauer 2006].

Dans le travail présenté dans cette thèse, nous proposons des solutions à ces problèmes avec une approche basée sur les messages centrée sur les données pour le test de conformité. Dans notre travail, les événements dans une trace sont des *messages*, c.-à-d. des rassemblements des champs de données structurés, avec une

partie de contrôle définie en fonction des données[2]. Une définition formelle d'un message est fournie et des fonctions pour le traitement des valeurs des champs de données également. Des observations spécifiques sont définies par des contraintes ou restrictions sur les messages, par exemple que certains champs de données dans le message contiennent une valeur ou un intervalle des valeurs, et les relations à observer entre messages multiples sont également définies de cette manière, ex. que les champs de données entre deux messages correspondent (ou ne correspondent pas) selon leurs valeurs. Ces restrictions sur les messages sont définies sous la forme de clauses de Horn, qui présente l'avantage de permettre la réutilisation des clauses.

Des relations temporelles entre des événements sont définies par la quantification ($\exists$, $\forall$) sur les messages, et la direction de recherche (avant, arrière) est spécifiée par des relations d'ordre explicites, ex. $\forall_{x<m}$: *condition* indique que *condition* doit être vraie pour chaque message apparaissant avant m dans la trace. Cela permet non seulement de définir des événements forward et backward, mais aussi un mélange des deux. Puis la syntaxe et la sémantique pour la logique utilisée pour exprimer les propriétés et un algorithme pour l'évaluation des propriétés sur des traces offline sont définies. L'algorithme permet d'observer des occurrences multiples d'une propriété dans une trace et retourne une valeur de vérité dans $\{\top, \bot, ?\}$ (pass, fail ou inconclusive) pour l'évaluation de la propriété. Un résultat '$\top$' indique que la propriété est satisfaite sur la trace, un résultat '$\bot$' indique que la propriété n'est pas satisfaite et '?' indique que la satisfaction de la propriété ne peut pas être établie, la fin de la trace étant atteinte.

En utilisant cette sémantique, le problème de tester les propriétés sur des traces finies devient le problème de déterminer si un verdict **fail** ou un verdict **inconclusive** devrait être donné comme résultat de satisfaction '?'. En d'autres termes, est-ce que le défaut d'observation d'un comportement attendu signifie que le comportement n'a jamais été produit? (trace suffisamment longue?). Quatre solutions alternatives sont proposées dans notre travail pour faire face à cette question.

1. Supposer que la trace n'est jamais assez longue, c'est-à-dire, seulement des verdicts **inconclusive** peuvent être fournis si un resultat '?' est observé. Bien que cette hypothèse est très stricte, et peut ne pas fournir d'informations utiles dans la plupart des cas, une analyse plus approfondie des verdicts le pourrait. Par exemple, si le nombre et la distribution des verdicts **inconclusive** sont significatifs, cela peut être un indice d'une faute dans l'implémentation.

2. Supposer que la trace est toujours suffisamment longue, c'est-à-dire, que chaque résultat '?' est une indication d'une défaillance. Si des traces avec un grand nombre de messages sont utilisées, cela peut être une solution acceptable. Toutefois, certains résultats faux positifs peuvent être produits aux bords de la trace.

[2]Ceci est inspiré, bien sûr, par des traces réelles, où les événements sont des paquets (packets).

3. Préciser un comportement alternatif, conditionnel à observer. Si lors de la tentative d'observer une propriété particulière, un comportement conditionnel est observé en premier, alors un verdict **fail** est rendu, sinon, un verdict **inconclusive** est rendu. Ceci est similaire à identifier l'état initial ou final d'une spécification dans la trace dans des autres approches basés sur invariants. Malheureusement, la condition pourrait ne pas exister, ou il pourrait être difficile à définir, car la causalité avec les critères dans la propriété attendue doit être spécifiée.

4. Définir explicitement un comportement qui ne doit pas être observé, ou comportement négatif. Comme la question est de déterminer quand l'absence d'observation d'un comportement attendu est indicative d'une défaillance, il n'y a pas de problème pour détecter quand le comportement a effectivement eu lieu. Si un comportement négatif est défini comme une propriété, donc la satisfaction de la propriété (un résultat '$\top$') doit produire un verdict **fail**. Cependant, le comportement négatif peut ne pas être toujours facile de définir à partir des exigences du service.

Avec ces solutions alternatives une nouvelle définition des invariants est fournie dans notre travail. Des invariants sont définies comme une paire $(test, condition)$. Le test est la propriété effective qui doit être observée dans la trace et la condition est une observation alternative qui permet de déterminer si un résultat '?' par l'algorithme est une réelle défaillance dans l'IUT ou qu'aucun verdict ne peut être produite, ce qui offre une solution possible à la question précédemment décrit. Deux types d'invariants sont définis: *positifs* et *négatifs*, pour tester des séquences qui doivent être observées dans la trace, mais aussi pour celles qui ne devraient jamais être observées.

Une procédure pour fournir des verdicts de conformité (**pass**, **fail**, **inconclusive**) sur une paire $(test, condition)$ d'un invariant est fournie ainsi.

Nous avons également mis en oeuvre les concepts décrits dans un prototype, afin de montrer l'applicabilité du travail et pour tester les algorithmes sur des traces IMS réelles.

Organisation du manuscrit

Le présent manuscrit de thèse est organisé comme suit:

1. Dans le deuxième chapitre, nous présentons un état de l'art des techniques de test de conformité. Nous allons des concepts généraux de test de conformité, méthodes formelles et le test formel de conformité, à un bref aperçu des techniques de test actif et une vue plus détaillée des approches de test passif pour la conformité. Nous fournissons également un aperçu général du runtime

verification en tant que discipline, ses objectifs, certains travaux pertinents à l'approche présentée dans cette oeuvre ainsi que la relation du runtime verification avec le test passif.

2. Dans le troisième chapitre, nous présentons SIP et l'IMS. On commence par un aperçu du SIP, ses entités et leur comportement, ainsi que la syntaxe des messages et des données pertinentes menées par les messages SIP. Pour l'IMS, nous décrivons brièvement quelques-unes de ses entités centrales ainsi que certains des comportements de deux services IMS, le Push-to-talk Over Cellular et le service Presence.

3. Dans le quatrième chapitre, nous présentons notre approche initiale pour le test de conformité des services l'IMS, par une étude de cas industrielle. L'approche sert à décrire quelques-unes des limites du test avec des propriétés centrées sur parties de contrôle et fournit une partie de la motivation pour le travail présenté dans cette thèse.

4. Le cinquième chapitre contient notre contribution principale. Nous détaillons d'abord les limites des approches des invariants actuelles, à travers des questions sur la causalité des entrées et sorties, ainsi que les exigences d'une approche centrée sur les données. Ensuite, nous commençons le détail de chaque partie de la contribution: la définition des traces comme une séquence des messages et la formalisation des messages comme des structures de données, la description de la syntaxe et la sémantique des formules, tant pour les clauses basées sur Horn et l'établissement de relations temporelles. Puis nous décrivons l'algorithme pour l'évaluation des formules dans les traces, et nous détaillons sur la complexité de l'algorithme. Finalement, nous détaillons la procédure d'évaluation des invariants positifs et négatifs et l'évaluation de leur partie de test et condition. Nous finissons par donner un exemple de définition d'un service basé sur SIP, ainsi que des expériences sur des traces réelles du protocole.

5. Sur le dernier chapitre, nous concluons la présentation de notre travail, en fournissant un résumé de nos contributions, ainsi que d'un ensemble de perspectives pour des extensions futures et des améliorations pour notre approche, en particulier en termes d'évaluation en temps réel et de l'évaluation des propriétés en cours de l'exécution.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 General Context

In current times, when communication is essential and an immense array of services is available online, computer networks continue to grow and new communication protocols and services are continuously being developed. Regarding this area, communication standards are essential to enable interworking of systems and correctness of the standard definitions and their implementations is fundamental to ensure that platforms can communicate. While formal *verification* techniques are used to evaluate correctness of standards [Woodcock 2009], the conformance of their implementations to the standard requirements is usually evaluated through *testing* techniques.

Conformance testing is the process of checking that a system possesses a set of desired properties and behaves in accordance with some predefined requirements. There is a high level of consciousness of its importance and impact for the future deployment and use of software and systems. This is notably observed with the numerous works tackling the testing areas; works provided by the research communities of course [Hierons 2009] but also by the industry [ETSI/ES 201 873-1 2007] and the standardization institutes [ETSI/ETR 022 1993, ISO/IEC 9646 1994].

An important subset of testing approaches is based on *formal methods*. These methods rely on *formal specifications*, mathematical models constructed using the informal requirements of the system during the specification phase of the development. The formal specification is used to support the testing process, for instance, with automatic test case generation techniques and specification coverage metrics, it is possible to experimentally and reliably determine whether a protocol implementation satisfies its requirements. In a conformance testing context, *black-box* methods are generally used, where the internal structure of the implementation is unknown and all evaluation is done through observation of inputs, outputs and environment of the *implementation under test* (IUT). Two main mechanisms are usually distinguished for this purpose according to their level of interaction with the IUT: *passive* and *active* testing.

Active testing is based on the execution of specific test sequences against an implementation under test. Test sequences are generated from a formal specification of the protocol according to different coverage and conformance satisfaction criteria. The tests may be generated automatically or semi-automatically based on various

hypotheses about the implementation and different test goals. Test execution is performed through *points of control and observation* (PCOs), i.e. execution interfaces. These PCOs are defined in the context of a testing architecture, a particular distribution of testers that allows them to interact with the platform and communicate with each other if necessary.

Passive testing is based on the observation of input and output events of an implementation under test during runtime. The term "passive" means that the tests do not disturb the natural operation of the system as the implementation is not stimulated. The record of the event observation is called a *trace*. In order to check the conformance of the IUT, this trace will be compared to a set of expected behaviors, defined either by a formal model (whenever available) or by one or more functional (conformance) properties. Passive testing provides an alternative to active testing, whenever the requirements of the latter (definition of testing architectures and access to the system interfaces for controlling the inputs) are unfeasible or undesired. Passive testing also provides a useful alternative when the implementation cannot be shutdown or stopped for a long period of time.

Within passive testing, *invariant-based testing* methodologies can be distinguished. There, a set of (critical) conformance properties is defined, either automatically from a formal specification, or manually from the system's requirements. Such properties specify sequences of events that must be observed in the trace to establish conformance of the implementation. The principle is based on the *trace preorder* implementation relation, that is, if a trace is observed from the IUT that cannot be produced by the specification, then this is indication of a fault in the implementation. In invariant-based testing, the behavior that must be observed is defined by the conformance properties (the invariants).

Invariant-based testing methodologies have some advantages over specification-based passive techniques. Passive testing techniques compare the observed behavior (in the trace) with the expected behavior from the specification, in order to determine whether the former can be produced by the latter. This requires verification of every state in the specification in the worst case. Invariant methodologies provide a method for rapid determination of conformance to critical properties. These techniques can also be useful for testing when the specification is not available, which is often the case for large systems. Finally, invariant-based techniques provide interesting perspectives for monitoring conformance properties during the execution of the IUT. In this last context, a related area has gained popularity in the last decade from the verification community, called *runtime verification* [Leucker 2009].

Runtime verification is a discipline, derived from *model checking*, that deals with the study of verification techniques that allow checking whether a run of a system satisfies a given correctness property. Differently from passive testing, execution traces in runtime verification are not limited to input/output events and are generally described as a set of states of the system. As with model checking, runtime verification deals to a great extent with the technical aspects of property evalua-

tion and monitor generation. Nevertheless, many of the techniques and concepts of runtime verification may also be used for passive conformance testing.

1.2 Contributions

The Internet Multimedia Subsystem (IMS) is a standardized framework for delivering IP multimedia services to users in mobility. It provides a centralized architecture for telecommunication operators to integrate and provide access to multiple 3G voice and multimedia services, while also providing Quality of Service (QoS) control and charging features for operators. IMS services rely heavily on IETF standards, in particular the Session Initiation Protocol (SIP), for their session control and communication.

Differently from other protocols, the SIP specification [Rosenberg 2002] does not describe a service, but a set of extensible primitives for session establishment, configuration and termination of services, with over 200 related standards[1] between extensions and complements. IMS services usually integrate multiple of these extensions, thus providing interesting challenges in terms of conformance testing.

Conformance testing is generally about establishing compliance of a particular protocol implementation to its standard, however, some of the techniques of conformance testing may also be applicable for testing of service implementations. In the case of IMS services, features from multiple protocol and service specifications are integrated, and their integration described in standardized documents [Open Mobile Alliance 2006, Open Mobile Alliance 2010]. Conformance testing in the context of IMS services then must consider two separate aspects: *1)* conformance of each protocol and extension implemented by the service to its particular specification, *2)* and conformance of the integrated extensions to the requirements of the service.

Furthermore, lack of open implementations for IMS services, and usual lack of access to the operator's service interfaces restricts the possibilities for their active testing. In this work we provide improvements to passive testing techniques for the testing of IMS and SIP-based services, that we also believe may provide useful for other application-layer protocols and services.

Traditional passive testing techniques derive from model-based testing techniques, usually based on Finite State Machine (FSM), Extended FSM (EFSM) and Labeled Transition System (LTS) specifications. These models usually presume a causal relation between the control parts of inputs and outputs in the transitions of the model, due to their original use for reactive systems, meaning that traces (observations) from the system take the form of a sequence of input/output pairs. This allows testing techniques, and in particular invariant-based techniques, to make use

[1] For an up-to-date list the reader can refer to http://www.packetizer.com/ipmc/sip/standards.html

of such causality for defining properties, sequences of inputs/outputs, that must be seen on the trace to establish conformance of the implementation. For traces of many protocols, such causality is not always applicable, since many outputs can be expected for an input and vice versa. Moreover, in collection of real traces, particularly for centralized services, the trace may contain interactions with multiple clients, which makes establishing causality based on control parts even more difficult to determine.

In such type of traces, causality between events in a trace can many times be established only through data parts of messages. However, as traditional testing (and verification) approaches derive from works with finite state (or labeled transition) models, they are usually propositional in nature, meaning that they first take into account control parts, with data parts as an extension of control parts, generally in the form of parameters added to the control parts. This means a reduced expressiveness of formulas to establish data relations or expressiveness in expense of succinctness of formulas. This becomes even more critical when constraints other than equality of data parameters is needed. Although some works in invariant-based approaches have been proposed to deal with data in the form of constraints [Ladani 2005], they require the use of a specification to determine constraints, which limits the practicality of the approach, since a specification is not always available, particularly for large systems.

One last issue has to be considered when causality between events is removed. When testing with properties on finite traces, when testing a property such as "if event x happens then event y must be observed on the trace", it may occur that event x is observed, but event y is not observed. Unless some assumptions about the execution are made, it may not be possible to distinguish between the cases: "the event y was never produced by the implementation" and "the trace collection finished before y could be observed". This issue had already been noted for backward properties [Bayse 2005] ("if x is observed, then y must have been observed before") and a solution based on passive testing *homing phase* to detect if the initial state of the specification was contained in the trace was proposed. However, if a specification is not immediately available, such solution is not feasible, and no equivalent solution exists for forward properties. Similar issues have been identified in the literature of runtime verification [Bauer 2006].

In the work presented in this thesis, we propose solutions to these issues with a message-based/data-centric approach to conformance testing. In our work, events in a trace are *messages*, i.e. collections of structured data fields, with control part defined as a function of the data[2]. A formal definition of a message is provided and functions for dealing with data field values are defined. Specific observations are defined through constraints or restrictions on messages, e.g. that some data field in the message contains a specific value or range of values, and relations to be observed between multiple messages are also defined in that way, e.g. that data fields

[2]This is inspired of course, by real traces, where events are packets.

between two messages match (or do not match) in their values. These restrictions on messages are defined in the form of Horn clauses, which has the added benefit of allowing re-usability of clauses.

Temporal relations between events are defined through quantification ($\exists, \forall$) over messages, and search direction (forward, backward) is specified by explicit order relations, e.g. $\forall_{x<m} : \mathit{condition}$ indicates that *condition* must be true for every message appearing before m in the trace. This allows not only to define forward and backward events, but also a mix of both. Both the syntax and semantics for the logic used to express properties and an algorithm for evaluation of the properties on off-line traces are defined. The algorithm allows to observe multiple occurrences of a property in a trace and returns a truth value in $\{\top, \bot, ?\}$ (true, false or inconclusive) for the evaluation of the property.

Invariants are defined using these definitions as a pair $(\mathit{test}, \mathit{condition})$. The test is the actual property that needs to be observed in the trace and the condition is an alternative observation that allows to determine if an '?' result from the algorithm is an actual failure or that no verdict can be produced, offering a possible solution to the previously described issue. Two types of invariants are defined: *positive* and *negative*, to test for sequences that must be observed in the trace, but also for those that should never be observed. A procedure for providing conformance verdicts (**pass**, **fail**, **inconclusive**) on an invariant's $(\mathit{test}, \mathit{condition})$ pair is provided as well.

We have also implemented the concepts described in this work into a framework, in order to show the applicability of the work and tested the algorithms on actual IMS traces.

1.3 Thesis plan

This manuscript is organized as follows:

1. In the second chapter we present a state of the art of conformance testing techniques. We go from the general concepts of conformance, formal methods and formal testing for conformance, to a brief overview of active techniques and more detailed view of passive testing approaches for conformance. We also provide a general overview of runtime verification as a discipline, its objectives, some relevant works to the approach presented and the relation with passive testing.

2. In the third chapter we present SIP and the IMS. We begin by an overview of SIP, its entities and some of their behavior, along with the message syntax and the relevant data carried by SIP messages. For IMS, we briefly describe some of the entities in its core as well as some of the behavior of two IMS services, the Push-to-talk Over Cellular and the Presence service.

3. In the fourth chapter we present our initial approach for conformance testing of IMS services, through an industrial real-case study. The approach there serves to describe some of the limitations of testing with control-centric properties, and provides some of the motivation and inspiration for the present work.

4. The fifth chapter contains our main contribution. We first detail on the limitations of the current invariant approaches, through the issues of causality of inputs and outputs, as well as the requirements of a data-centric approach. We then begin the detail of each part of the contribution: the definition of traces as sequence of message events and the formalization of messages as data structures, the description of the syntax and semantics of formulas, both for the Horn-based clauses and the establishment of temporal relations. Then we describe the algorithm for evaluation of formulas in traces, and describe the complexity of the algorithm. Finally we detail on the evaluation procedure for positive and negative invariants and the evaluation of their test and condition parts. We finally provide an example definition for a SIP-based service, as well as experiments on real protocol traces.

5. On the final chapter we conclude the presentation of our work, providing a summary of our contributions, as well as a set of perspectives for future extensions and improvements for our approach, particularly in terms of real time evaluation and runtime evaluation of properties.

Chapter 2

State of the Art

Contents

2.1 Testing of communication protocols

Testing is the process of operating a system or component under specified conditions, to observe the results and provide an evaluation of such system or component [IEEE Std 610.12-1990 1990]. Many types of testing exist, depending on the property being evaluated, for instance, it is possible to test for performance, usability, scalability, etc. *Testing for correctness*, the main concern of the current work, is testing as a means of checking that a system's (or component's) behavior correspond with a predefined, desired behavior. More specifically, testing for correctness is the process of detecting *faults* or defects in a system implementation, either by identifying *errors* (incorrect internal states) during the execution of the system, or by the observation of an incorrect external behavior (a *failure*). Testing can only show the presence of faults and never their absence [Dijkstra 1970].

In this section we develop the concepts related to testing of communication protocols, or *conformance testing* and provide an overview of the literature regarding this area. Although this section intends to provide a general view of conformance testing methodologies, it will also detail particularly the subset of conformance testing known as *passive testing*, which directly relates to the work in this document.

2.1.1 Conformance testing

Protocol conformance testing is a branch of testing designed to determine compliance of protocol implementations to their standards. More precisely, the objective

of conformance testing is to determine compliance of a particular *implementation under test* to a set of *conformance requirements* defined directly or indirectly by the protocol standard documents. Although a distinction can be drawn with (general) *conformance testing*, many concepts are common and therefore through the rest of this work we will use both concepts as equivalents.

Protocol conformance testing is a type of *black-box* testing, or *functional testing*, meaning that the internal structure of the implementation is ignored (or unknown), and the evaluation is done based on the observation of inputs, outputs and execution conditions during the execution of the test (or during normal runtime in *passive testing*). In contrast, in *white-box testing* or *structural testing*, usually used in software testing, tests are derived taking into account the internal structure of the system or program. The independence between test and implementation in black-box testing is of great importance in removing biases from the testing process.

The first step in protocol conformance testing is identifying the conformance requirements from the standard document. The ISO/IEC standard for conformance testing [ISO/IEC 9646 1994] (a good overview is provided in [Tretmans 2001]), identifies two types of conformance requirements. *Static conformance requirements* define the capabilities and interdependence between capabilities to be supported by implementation of the protocol. *Dynamic conformance requirements* specify the observable behavior permitted and/or required by the protocol standard.

Static conformance requirements are used for capability testing. The Protocol Information Conformance Statement (PICS), a document provided by the developer stating the capabilities of the implementation, is reviewed along with the static conformance requirements to check consistency in the implemented capabilities in a process called the *static conformance review*.

Dynamic conformance requirements are used for behavior testing of the implementation. A *test case*, an individual definition of the input/output conditions required to satisfy a requirement, are derived from dynamic conformance requirements. The standard defines a several step process in order to go from requirements to executable tests. Although intermediate steps are also defined, the three defined below describe in general terms the process.

1. One or more *test purposes*, are derived from each dynamic conformance requirement. A *test purpose* is a precise, natural language description of what needs to be tested in order to determine the satisfaction of a particular conformance requirement.

2. From the test purposes, a set of *abstract test cases* is defined. An *abstract test case* is a test case that specifies the conditions for satisfaction of a particular requirement in a way that is independent of the protocol implementation. Abstract test cases are written in some test notation, Tree and Tabular Combined Notation (TTCN) [ISO/IEC 9646 1994, part 3] being the

one recommended by the standard, which has since been updated in TTCN-3 [ETSI/ES 201 873-1 2007].

3. In the last step, an *executable test suite* is derived from the abstract test cases taking into account the particular implementation and environment that will be used.

Finally, tests are executed individually in the implementation, which in this step is referred to as *implementation under test* (IUT). In the standard, different *testing architectures* are proposed, in accordance with the OSI network model. Each tester is placed in a *Point of Control and Observation* (PCO), located in the communication interfaces of the IUT, and depending on the architecture, coordination between testers is also laid out. The concept of *System Under Test* (SUT) is also defined for some architectures, where the SUT includes the IUT as well as the local testers. The execution of each test provides one of the following three possible *verdicts*: **pass**, meaning the implementation conforms to the test, **fail**, meaning that the implementation does not conform to the test and **inconclusive**, meaning that no definite conclusion can be drawn from the test.

An implementation *conforms* (in terms of ISO/IEC 9646) to the standard if the verdict for every test is a **pass**.

2.1.2 Formal testing

Testing techniques are applicable in every stage in the software development process (specification, design, code), and particularly when used together with *formal methods*, they can help identifying problems much earlier in the development, when the cost of modification is low. *Formal methods* [Hierons 2009] provide a means not only to verify the results during the system definition, but can also support the testing process. Figure 2.1 shows some of the possible relations between formal methods and testing techniques at different development stages.

The methodology used by formal methods starts with the *specification* phase, where the informal requirements for the system are used to construct a *formal specification* of the system behavior, a mathematical model written using a *formal specification language*. Different types of languages exist, depending on the paradigm used (model-based, process algebra-based, etc.). *Finite state-based languages*, as those based in FSM and EFSM [Lee 1996], e.g. SDL [ITU-T Z.100 1999], are particularly popular with conformance testing of protocols, given the similarities between the FSM models and the control structure of a protocol [Hierons 2009].

The formal specification of the system is used to guide the implementation process. The implementation can be done through successive refinements of the specification, or by direct coding. The specification is usually used in the *verification* process, where the code is checked against the specification in order to determine satisfaction. The specification itself can also be *verified*, by checking that specific

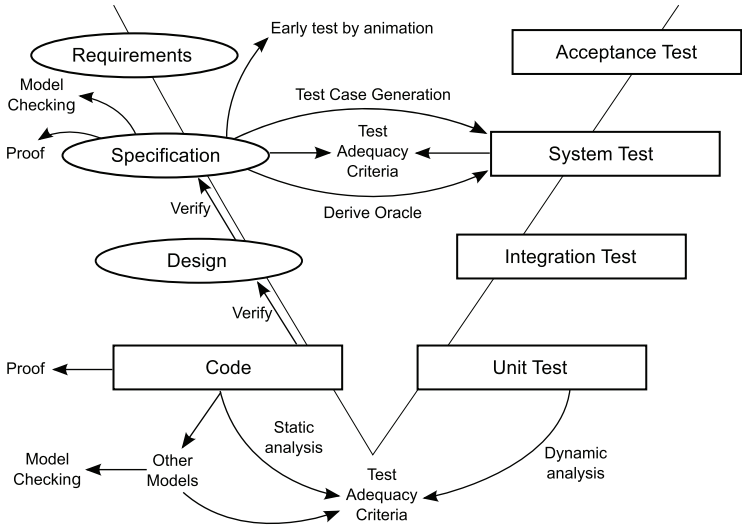


Figure 2.1: Formal methods and testing.

critical properties of the system are included. Verification should not be confused with *validation*. While the former deals with satisfaction of a product of a development phase with respect to a previous phase, the latter deals with satisfaction of the finished product, the system, with the intended behavior.

The specification can finally be used to support the testing process. Assuming (or having verified) that the specification is correct with respect to the requirements, testing in a formal context means simply experimentation to validate that the implementation supports all the behavior defined by the specification. In other words, to determine *conformance* of the implementation to the formal specification.

2.1.3 Conformance testing with formal specifications

Protocol conformance testing with formal specifications follows a similar methodology to the one described in Section 2.1.1. A formal specification is defined in accordance to the protocol requirements and used during the testing phase in order to determine the conformance of an IUT. The meaning of *conformance*, between a given specification `S` and an implementation `I`, is critical for the understanding of the area and as such it is discussed below.

The most general definition of conformance is provided in [Tretmans 1999], and we reproduce it here. Conformance relates to specifications and implementations. The universe of specifications is defined by the set $SPECS$ and the universe of all IUTs is denoted by $IMPS$. Considering this, conformance is defined as the relation

$$\textbf{conforms-to} \subseteq IMPS \times SPECS$$

where given an IUT `IUT` $\in IMPS$ and a specification `S` $\in SPECS$,

$\texttt{IUT}$ **conforms-to** $\texttt{S}$ expresses that $\texttt{IUT}$ is a correct implementation of the specification $\texttt{S}$.

In order to relate real implementations (in $IMPS$) with specifications, the assumption is made that every IUT $\texttt{IUT} \in IMPS$ can be modeled by a (possibly unknown) formal object $\texttt{I}_{\texttt{IUT}} \in MODS$, where $MODS$ is the universe of formal objects. This assumption is known as a *test hypothesis* [Bernot 1991, Gaudel 1995]. Under this assumption, an *implementation relation* is defined between models and specifications as $\textbf{imp} \subseteq MODS \times SPECS$. An implementation $\texttt{IUT} \in IMPS$ is said to conform to an specification $\texttt{S} \in SPECS$ if and only if the model of the implementation $\texttt{I}_{\texttt{IUT}} \in MODS$ is implementation related with $\texttt{S}$

$$\texttt{IUT}\ \textbf{conforms-to}\ \texttt{S} \Leftrightarrow \texttt{I}_{\texttt{IUT}}\ \textbf{imp}\ \texttt{S}$$

Different implementation relations can defined, depending on the model or specification language used, an overview of those most relevant to our work is provided below.

2.1.3.1 Equivalence relations

The most general type of implementation relation is *equivalence*, traditionally used for testing with state-based specifications, in particular *Finite State Machines* (FSM). FSMs can be *deterministic* (DFSM) or *non-deterministic* (NDFSM), however we will use the general term FSM to denote a deterministic finite state machine.

Definition 2.1.1. A *Finite State Machine* FSM is a 6-tuple $(S, s_0, I, O, \delta, \lambda)$, where

- S is a finite set of states, with $s_0 \in S$ the initial state
- I is a finite set of input symbols
- O is a finite set of output symbols
- $\delta : S \times I \to S$ is the state transition function
- $\lambda : S \times I \to O$ is the output function

When the machine is in a current state $s \in S$ and receives an input $i \in I$ it moves to the next state specified by $\delta(s, i)$ and produces the output specified by $\lambda(s, i)$.

The FSM of an IUT $\texttt{I}$ conforms to a specification $\texttt{S}$ if $\texttt{I}, \texttt{S}$ are *equivalent* [Lee 1996], meaning that they have the same number of states and transitions (they are *isomorph*). In terms of external behavior, and under the assumption that both FSMs have the same number of states, $\texttt{I}$ and $\texttt{S}$ are equivalent if they cannot be distinguished by any sequence of inputs, i.e. both the specification $\texttt{S}$ and the implementation $\texttt{I}$ will generate the same outputs for identical input sequences.

This type of equivalence is also called *trace equivalence*, where a *trace* is a sequence of input/output pairs. For NDFSMs, other types of conformance relations exist, including equivalence (although with a different definition), quasi-equivalence and reduction relation [Bochmann 1994].

In testing with process algebra specifications, inputs and outputs are generally not distinguishable, and non-observable transitions are possible. Here, a different set of equivalences can be used as implementation relations, depending on the type of observations that are possible during testing. A commonly used formalism in this area are *Labeled Transitions Systems* (LTS), that we define below.

Definition 2.1.2. A *Labeled Transition System* is a 4-tuple (S, Σ, T, s_0), where

- S is a countable, non-empty set of states
- Σ is a countable set of observable actions
- $T \subseteq S \times \Sigma \cup \{\tau\} \times S$ is a transition relation, and $\tau \notin \Sigma$ is an unobservable action
- s_0 is the initial state

A transition between states $s, s' \in S$ and action $a \in \Sigma$ is denoted as $s \xrightarrow{a} s'$. Given a sequence of actions $\sigma \in \Sigma^*$, $s \stackrel{\sigma}{\Rightarrow} s'$ denotes the set of transitions taking from s to s' after applying the sequence σ, including transitions with unobservable actions.

The *trace equivalence* relation between LTS, requires the trace sets of two models $\mathtt{I}$ and $\mathtt{S}$ to be identical, denoted as $traces(\mathtt{I}) = traces(\mathtt{S})$[1]. A stronger version is *testing equivalence* [De Nicola 1984], defined as equality between observations during testing, denoted as $runs(t, \mathtt{I}) = runs(t, \mathtt{S})$ for every test t. The term *observation* in LTS considers tests that end in a lock (dead or alive), as well as successful tests, making it a stronger relation than trace equivalence. Other types of equivalence relations are also possible, as *bisimulation equivalence*, *observation equivalence* and *failure equivalence* [Fernandez 1991].

2.1.3.2 Other implementation relations

Other types of relations different from equivalence are also shown in the literature to be suitable as implementation relations for testing and test case generation. The work of J. Tretmans [Tretmans 1992] for LTS is fundamental in this area. *Preorder* relations are useful when it is only desired to express inclusion or ordering between models. *Trace preorder*, is a weaker requirement than trace equivalence, where an implementation $\mathtt{I}$ has a trace preorder implementation relation with a specification $\mathtt{S}$, denoted by $\mathtt{I} \leq_{tr} \mathtt{S}$ if and only if $traces(\mathtt{I}) \subseteq traces(\mathtt{S})$. *Testing preorder* ($\leq_{te}$)

[1] For $\mathtt{S} = (S, \Sigma, T, s_0)$, $traces(\mathtt{S}) = \{\sigma \in \Sigma^* | s \stackrel{\sigma}{\Rightarrow} s', \forall s, s' \in S\}$

is defined in a similar way with respect to tests, where observations about the implementation have to be a subset of observations about the specification.

Non-preorder relations are also described in the literature. The authors of [Brinksma 1988] define the *conf* relation for LTL, which improves on the testing preorder relation by restricting the domain (or language) of the tests to the one defined by the specification. The *ioconf* relation [Tretmans 1996b], extends the *conf* relation to deal with Input Output Transition Systems (IOTS). Other extensions for real-time systems [Krichen 2004] and symbolic systems [Frantzen 2004] have also been proposed.

2.1.3.3 Implementation relations and test generation

Implementation relations guide the testing process by providing formal conformance satisfaction criteria for an implementation under test. Evaluation of such criteria can be done either by *actively* controlling the IUT through test execution, or through *passive* observation of the normal operation of the IUT. These two methods are usually distinguished as **active testing** and **passive testing**, the latter of which will be discussed in detail in the next section, since it concerns the current work. An overview of active techniques is provided below.

Active testing techniques rely on test execution for conformance evaluation, and through the use of formal specifications, automated *test generation* methods are possible. The problem of conformance testing is reduced to finding a *test sequence* or *test suite* that allows to determine whether an IUT conforms with the specification w.r.t. a particular implementation relation. If the IUT *passes* the test suite, is declared conformant, otherwise, it is declared non-conformant. Figure 2.2, shows the traditional active testing methodology.

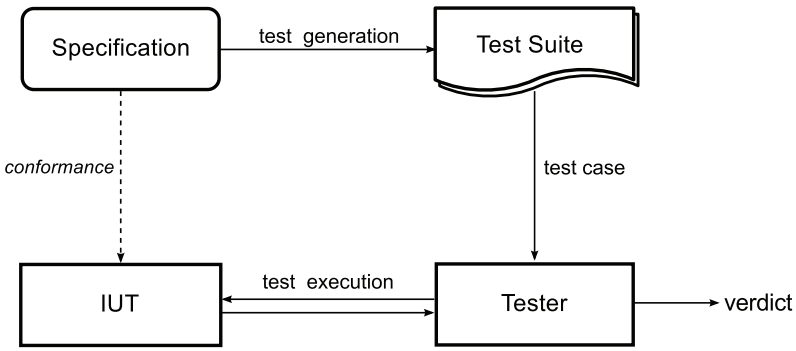


Figure 2.2: Active testing methodology.

For FSM based testing, the most used implementation relation is equivalence, which in the context of FSMs means *isomorphism*, i.e. equivalence of states and transitions. Two types of errors are possible under this definition of equivalence. **Output** errors occur when the machine returns the wrong output to a given input.

Transfer errors occur when the machine is left in the wrong state after executing a transition. Test case generation techniques for FSM-based specifications, attempt to characterize states after the execution of a test. Different methodologies have been developed, depending on different assumptions about the machine. State characterization through distinguishing sequences [Hennine 1964], characterization sets (W-method) [Chow 1978], unique input/output sequences (UIO-method) and transition tours [Naito 1981], are some of the techniques available in the literature. Detailed reviews of the different methodologies are provided in [Ural 1992, Lee 1996].

Test generation methods for the *conf* implementation relation [Tretmans 1996a], and the *ioco* implementation relation [Brinksma 1997] have also been developed.

2.1.4 Passive testing for conformance

Passive testing, as previously defined, is based of the observation of inputs/outputs to the IUT during normal runtime. Although it has some disadvantages with respect to *active* techniques, the most important being lack of control of the execution, it also has important advantages, particularly the ability to test an implementation without disturbing normal operation and in its natural environmental conditions. It is also the only option available under some circumstances, when the interfaces of the system are inaccessible, or when active testing is impractical due to the complexity of the system. *Online* and *Offline* approaches can be distinguished. In the former, the tester attempts to detect a fault during the execution of the system. In the latter, the evaluation of the system is done in recorded *traces*. A *trace* is a record of input/output events of the IUT. Figure 2.3 shows the traditional passive testing methodology. A description of some of the most important works and methodologies in this area are provided in the rest of the section.

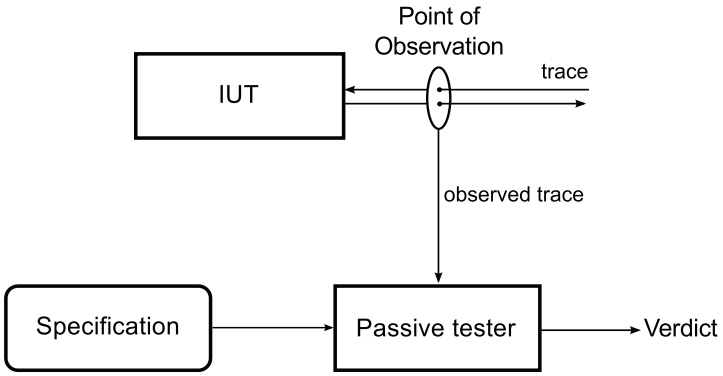


Figure 2.3: Passive testing methodology.

2.1.4.1 Testing with FSMs

One of the earliest works regarding passive testing for protocol testing is provided in [Lee 1997] in order to detect network faults. In their work, the network is modeled

by an FSM and *observational equivalence* is used as a conformance relation. Observational equivalence considers the implementation `I` as faulty if there is a behavior that cannot be observed in the specification `S`. Observations of the implementation are input/output pairs (a trace), each one assumed to represent one transition. A two stage algorithm is defined:

1. The passive *homing sequence* stage is designed to determine the state of the implementation at the beginning of the trace. At the beginning all states in `S` are considered candidates. The homing stage rules out one by one the states from which the next input/output pair in the trace cannot be produced, starting by the beginning of the trace. At the end of the stage we either end up with one candidate, which provides the starting point for the second stage, or we rule out all states, in which case, there is a fault in the implementation and testing ends.

2. The second stage, or *fault detection* stage, continues the comparison between the observed behavior (the trace), and the specified behavior. If an observation does not match the expected behavior, then a fault has been found, if the end of the trace is reached, no verdict can be given.

2.1.4.2 Testing with Extended FSMs

Finite state machines are a useful mode for specifying simple systems, however they quickly become unpractical to use as the size of the system grows. *Extended Finite State Machines* (EFSMs) provide a generalization of FSM with the incorporation of variables, predicates (conditions) and actions over the variables, allowing for more succinct specifications. The EFSM model provides the semantics for the Specification and Description Language (SDL) [ITU-T Z.100 1999].

Definition 2.1.3. An *Extended Finite State Machine* (EFSM) is a 6-tuple $(S, s_0, I, O, \vec{x}, T)$, where

- S is a finite set of states, with $s_0 \in S$ the initial state
- I is a finite set of input symbols with or without parameters
- O is a finite set of output symbols with or without parameters
- $\vec{x}$ is a variable vector
- T is a finite set of transitions

Each transition $t \in T$ is a 6-tuple $(s_t, f_t, i_t, o_t, P_t, A_t)$ where

- $s_t \in S$ is the beginning state of the transition
- $f_t \in S$ is the ending state of the transition

- $i_t \in I$ is the input for the transition
- $o_t \in O$ is the output of the transition
- $P_t(\vec{x})$ is a predicate on the variable values
- $A_t(\vec{x})$ is an action on the variable values

When the machine is in a current state $s \in S$ with variable values $\vec{x} = \vec{x}_s$ and receives an input i, it will follow the transition (s, f, i, o, P, A) if $P(\vec{x}_s)$ holds. In such case the machine will output o, update the variable values by the action $\vec{x} := A(\vec{x}_s)$ and leave the machine in state f.

Notice that input and output symbols can also contain parameters, in which case the parameters are represented by a vector $\vec{y}$. For a given transition, the input, output, predicate and actions may depend on the parameters, denoted as $i(\vec{y})$, $o(\vec{y})$, $P(\vec{x}, \vec{y})$, $A(\vec{x}$ and $\vec{y})$.

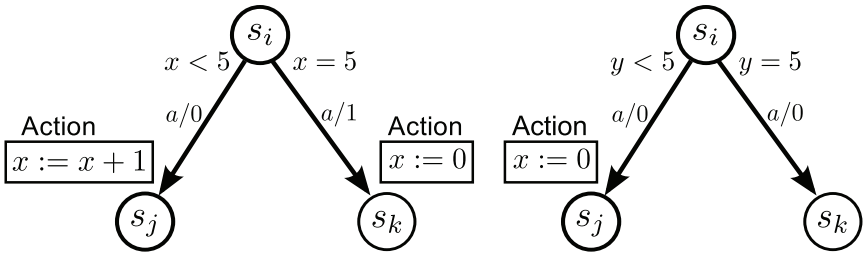
When testing with EFSM-based methods, two dimensions are distinguishable in the model. The **control portion** is defined by the observable behavior of the model. The **data portion** is defined by the variable and parameter values. Although an EFSM can be converted into an equivalent FSM for testing, doing so usually causes the *state explosion problem*. Testing with EFSMs then, must test both control and data portions.

Passive testing by value determination The work in [Tabourier 1999], provides one of the first approaches for passive testing with EFSM specifications. Figure 2.4(a) provides an illustration for the principle behind the algorithm. Assuming that the current state s_i is known and the value of variable x unknown, a similar procedure as with the previous FSM approach can be used. Upon observation of transition $a/1$, it can be deduced that the machine ends in state s_k and that x becomes $x = 0$. The new value of x can be used later in a predicate to determine if a transition should be fired.

It can occur, however, a case like the one shown in Figure 2.4(b), where the value of x is already known (suppose $x = 3$) and the value of y is unknown. Upon observation of $a/0$, both transitions are followed. The left side results in $x = 0$ and y unknown, the right side results in $y = 5$ and $x = 0$, however, since both transitions have the same I/O pair (evaluation is nondeterministic), the algorithm cannot decide what value to assign to the variables, and therefore x becomes undefined.

The algorithm is also divided in a homing phase and a fault detection phase. The homing phase follows the two rules illustrated before

1. For a given I/O pair, if several candidate transitions are possible, leading to different values for a variable, then the variable stays (or becomes) undefined.



(a) Variable value determination.

(b) Loss of information due to non-determinism.

Figure 2.4: Alternative cases in testing by value determination

2. If a variable appearing in a predicate is unknown, then the predicate is ignored and only I/O observations are used to decide on transition execution.

The homing phase finishes when an unique state is determined and all variable values are known. The fault detection phase evaluates the rest of the trace attempting to detect an erroneous I/O behavior with respect to the specification.

Passive testing by interval determination In the homing phase of the previous algorithm, variable values are calculated by using the actions, whenever a transition can be fired deterministically, or when the value determined by alternative actions is identical. The work in [Lee 2002] (and its extended version in [Lee 2006]) improves on the previous work by also including implicit information from predicate constraints, and keeping track of constraints and candidate values. Three resources are used for value determination.

- **Intervals** are maintained about integer variables, in order to keep record of the possible variable values. An interval about a variable v is denoted as $R(v) = [a, b]$, indicating that $a \leq v \leq b$. Information in the predicates is used to refine the interval. Intervals are used to determine the satisfaction of predicates later in the evaluation. When the value of variable v is found, the interval is represented as $R(v) = [a, a]$, for $v = a$, in which v is said *decided*.
- **Assertions** keep record of the constraints on variables. An assertion, denoted as $assert(\vec{x})$ is a boolean formula which contains the record of the constraints on variables up to the current point in the evaluation. When a transition is fired, the predicate is assumed to be true and added to the $assert(\vec{x})$. Actions with unknown right side values are also used to maintain the list of assertions (e.g. after an action $x_1 := x_2$ all references to x_1 can be replaced by x_2). During the process the consistency of the predicates in the assertion is confirmed to discard states and valuations.
- **Candidate Configuration Sets (CCS)** represent possible statuses of the machine. A candidate configuration set is represented by the triple

$(s, R(\vec{x}), assert(\vec{x}))$, where s is a state, $R(\vec{x})$ is the set of intervals for the different variables and $assert(\vec{x})$ is an assertion on $\vec{x}$.

The algorithm maintains a list of CCS at any point in the evaluation of the trace. The list starts empty and during the observation of a new event e in the trace, a new list of candidates is created with all possible machine statuses. We denote the current list of candidates as L_C and the list of candidates to evaluate in the next trace event as L_N. Each candidate $(s,\ R(\vec{x}), assert(\vec{x}))$ is evaluated according to the following criteria.

- A transition with event e is possible from state s.
- If the predicate can be evaluated, the evaluation of the predicate holds. Otherwise, the predicate needs to be consistent with $R(\vec{x})$ and $assert(\vec{x})$.

If the criteria are fulfilled, a new CCS is created for each new possible state and added to L_N. The actions are executed and the interval is refined to calculate the $R(\vec{x})$ and $assert(\vec{x})$ for each new candidate. This procedure is repeated for every candidate in L_C. The algorithm ends when there is only one candidate left, at which point the fault detection phase starts.

Other methodologies The authors of [Alcalde 2004], provide a similar methodology to the one described above, however in their approach, they follow the trace backwards in order to obtain a set of CCS at the beginning of the trace. In a second stage, they detect faults by attempting to find a path from the initial state of the EFSM that leads to one of the candidate sets at the beginning of the trace (the end of stage 1). If none are found, then it means there is a fault in the specification. An application to the Simple Connection Protocol is also evaluated.

In [Benharref 2007], the backwards and forward methods are combined for *online* passive testing of web services. Once an input or output appears in the trace, the algorithm attempts to find a set of candidates in the past of the trace that match the observed event. That information is then used for detection of faults, using the forward approach, upon reception of new events.

2.1.4.3 Invariant-based passive testing

The passive testing techniques discussed earlier are all based on comparison of the observed behavior (in the trace) with the expected behavior from the specification. These approaches require, in the worst case, to verify every state and transition in the specification for each trace evaluated, leading to low performance issues. The work in [Cavalli 2003] presented a first work on *invariant-based testing*. The principle is the following: *i)* from the specification (EFSM is used in the work), sequences of I/O pairs are extracted, *ii)* sequences are selected because of their uniqueness, i.e.

only one such sequence can occur in the specification (hence invariant). *iii)* If an invariant does not appear integrally on the trace (before the end is reached), then a fault has been detected.

An overview of the different invariant-based testing approaches is provided as follows.

Input/output invariants An I/O invariant consists of two parts, a *preamble* and *test*. The *preamble* is a sequence of events that needs to be found on the trace before the *test* can be evaluated. Three types of such invariants are defined in [Cavalli 2003].

- *Output invariants* allow to express properties such as 'immediately after the sequence *preamble*, the **output** *test* must be observed'. The following are examples of output invariants
  - $\underbrace{i_1}_{preamble} / \underbrace{o_1}_{test}$ denotes the property: "each time that input i_1 is observed, then the output o_1 must be also observed"
  - $\underbrace{i_1/o_1, i_2}_{preamble} / \underbrace{o_2}_{test}$ denotes the property: "each time that the pair i_1/o_1 and the input i_2 are observed, then the output o_2 must be also observed"
- *Input invariants* describe properties such as "immediately before the sequence *preamble*, the **input** *test* must be observed". Some examples of input invariants are
  - $\underbrace{i_1}_{test} / \underbrace{o_1}_{preamble}$ denotes the property: "the output o_2 must always be preceded by input i_1"
  - $\underbrace{i_1}_{test} / \underbrace{o_1, i_2/o_2}_{preamble}$ denotes the property: "the sequence $o_1, i_2/o_2$ must always be preceded by input i_1"
- *Succession invariants* describe more complex requirements such as those established by loops. For instance the sequence

$$\underbrace{i_1/o_1, i_1/o_1, i_1}_{preamble} / \underbrace{o_2}_{test}$$

  requires that the sequence i_1/o_1 is repeated twice before returning the output o_2. This would allow to test, for instance, a connection sequence, where two attempts are allowed before refusing.

Simple and Obligation invariants The work in [Arnedo 2003] extends the concept of invariants by introducing *simple invariants*, which are a generalization of the previously defined output invariants, allowing for wild-card characters to represent sequences of inputs or a single input/output. Since the most complex operation in testing with invariants is the generation of the invariant from the specification, for the described work, invariants are defined manually first (from the requirements), and later verified in the specification. A verification algorithm is provided. A simple invariants are defined to be consistent with the FSM formalism (definition 2.1.1), as follows

Definition 2.1.4. Let $M = (S, s_0, \mathcal{I}, \mathcal{O}, \delta, \lambda)$ be an FSM. A sequence I is a *simple invariant* for M if the following two conditions hold

1. I is defined according to the following Extended Backus-Naur Form (EBNF):

$$I ::= i/O \mid *, I \mid i'/o, I$$

   where $i \in \mathcal{I}$, $i' \in \mathcal{I} \cup \{?\}$, $o \in \mathcal{O} \cup \{?\}$ and $O \subseteq \mathcal{O}$

2. I is correct with respect to M

The wild-card character '$*$' represents any sequence of input/output pairs, and the wild-card character '?' represents any single input or output. Notice that '?' is not allowed in the final pair of the invariant.

In [Bayse 2005], the authors provide an extension for the previous work, where *obligation invariants* are defined to allow the description of obligation properties ("*if Y is observed then X must have been observed before*"). Obligation invariants are defined below as well as an example for both types of invariants after that.

Definition 2.1.5. Let $M = (S, s_0, \mathcal{I}, \mathcal{O}, \delta, \lambda)$ be an FSM. A sequence I is an *obligation invariant* for M if the following two conditions hold

1. I is defined according to the following EBNF:

$$I ::= i/\overline{O} \mid *, I \mid i/o, I$$

   where $i \in \mathcal{I} \cup \{?\}$, $o \in \mathcal{O} \cup \{?\}$ and $\overline{O} \subseteq \mathcal{O}$

2. I is correct with respect to M

The set $\overline{O}$ denotes the *obligation* part of the invariant.

Example 2.1.1. The **simple invariant**

$$I = req\_connect/connected, *, req\_disconnect/\{disconnected\}$$

expresses the following property: "A *disconnect* should be seen every time a disconnection is requested (by *req_disconnect*) if a connection had been granted previously"

The **obligation invariant**

$$I = req\_connect/connected, *, ?/\overline{\{data\_sent\}}$$

expresses the following property: "Before data can be sent, a connection MUST have been successfully granted"

Data parts in invariant testing Most of the described invariant testing approaches are derived from the FSM formalism, meaning that they only take into account control parts. The first work discussed ([Cavalli 2003]) in addition to extraction of control sequences, it extract separately the constraint information from the involved transitions. In order to test the property then, the correct sequence must be found and the constraints must hold, otherwise a fault is declared.

The authors of the second work [Arnedo 2003] propose a small modification to the obligation invariant approach, in order to deal with constant data parameters. Such proposal is taken as a starting point by the authors of [Ladani 2005], where concept of simple and obligation invariants is extended to match the EFSM formalism. Invariants are defined manually from the requirements, and input/output events can contain parametric variables. The invariants are then verified in the specification in order to check correctness, but also to extract the required constraints that will be evaluated during the validation of the invariant.

2.2 Runtime verification

Although the *verification* and *testing* communities have usually dealt with different issues and through different methodologies, in the last couple of decades there has been increased work in using verification techniques for testing [Fraser 2009]. In the context of passive testing, a parallel can be drawn between invariant-based techniques and those of *runtime verification*.

Runtime verification is a discipline, derived from *model checking*, that deals with the study, development and application of verification techniques that allow checking whether a run of a system under evaluation satisfies or violates a given *correctness* property [Leucker 2009]. It is portrayed in the literature as a *lightweight* verification technique, dealing only with the aspects of the system that can be evaluated during runtime, in opposition to traditional verification, which deals with all possible runs of a system.

In general terms, the methodology for runtime verification is the following: the system or implementation is assumed to behave as some model, M, some part of which is available during runtime (states, transitions, variable values, etc). As with

model checking, satisfiability of a given correctness property ϕ, must be determined on the runtime observations of the visible part of the model (the *trace*). Verification of a property can be done either *online*, i.e. during system execution, or *offline*, i.e. in recorded traces.

Similar issues to those in model checking are dealt with in runtime verification: definition of suitable logics for expressing properties, satisfiability (or monitorability) and verification of properties. However, while model checking deals with infinite runs of the system, runtime verification deals with finite executions. This means that many aspects and techniques of the former cannot be directly applied to the latter.

Determining the satisfaction of a correctness property (i.e. the verification part), involves the creation of a *monitor*. The *monitor* incrementally reads the trace of the system and yields a *verdict*, usually in the form of a truth value in some range (e.g. $\{true, false\}$ or a probability $[0, 1]$). A big part of works in this area deal with the generation of monitors for different type of properties and systems. An overview of different works in this area is provided by the authors of [Delgado 2004].

As in model checking, correctness properties are usually written in some variant of LTL [Vardi 1996] as it can bee seen in the works by [Havelund 2002, Giannakopoulou 2001, Havelund 2003], however the semantics of the formulas and the methods for generation of monitors differ. In terms of semantics, a work relevant to ours, is presented in [Bauer 2006], where a three valued semantics (*true*, *false*, *inconclusive*) is introduced for evaluation of properties.

Definition 2.2.1. Let $u \in \Sigma^*$, where Σ is an alphabet, denote a finite trace. The truth value of an LTL formula ϕ w.r.t. u denoted by $[u \vDash \phi]$, is an element of $\{\top, \bot, ?\}$ and is defined as follows.

$$[u \vDash \phi] = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega\colon\ u\sigma \vDash \phi \\ \bot & \text{if } \forall \sigma \in \Sigma^\omega\colon\ u\sigma \nvDash \phi \\ ? & \text{otherwise} \end{cases}$$

Put into words, this indicates that a $\top$ result can be provided for a formula ϕ in a trace u, only if for every future extension of the trace $u\sigma$, the formula remains true. The set Σ^ω denotes the set of all infinite sequences that can be formed from the values in the alphabet Σ.

From the semantics defined in that work, it is easy to see that results are related to the evaluation of border cases. For instance, for an LTL formula $\mathbf{X}\phi$ (ϕ must hold in the next state), if the evaluation is performed at the end of the trace, the next state is unknown, therefore, the truth value of the formula can only be *inconclusive*. Only after the next state is observed (if evaluating in performed *online*), then the monitor can decide the value.

Interesting approaches related to the present work are those dealing with data. In [Stolz 2008], the concept of *parameterized propositions* is introduced. Proposi-

tions contain data variables and quantifiers can be defined by the introduction of a '$\rightarrow$' operator. A property "every opened file must be closed" is described by the formula $\mathbf{G}[\forall x : open(x) \rightarrow \mathbf{F}close(x)]$, where the formula can only be true if the temporal relation between events $\{open, close\}$ holds for every possible value of x. In [Halle 2008, Halle 2009], a logic for evaluation in message-based work-flows is defined. Here, data is a more central part of the definition of formulas and LTL temporal operators are used to indicate temporal relations between messages in a trace. A formula $\mathbf{G}[\exists_a x : x = \text{'A'} \rightarrow \mathbf{F}(\exists_b y : y = 200)]$, indicates that each time a message appears, where the value of the field a of the message is 'A', then a future message where the field b has a value of 200 must also be found. Some more details on these and other works, and their comparison with the approach presented in this thesis are provided in Chapter 5.

Another important aspect of runtime verification is monitorability of properties. The traditional classification of properties from verification into *safety*, *invariance* and *liveness* [Bérard 2001, part II], is not generally applicable in runtime verification, since the set of monitorable properties is smaller than in model checking. Issues related to monitorability of properties are discussed in [Bauer 2007a, Falcone 2010a, Falcone 2010b] and alternative classifications are proposed.

Finally, some differences and similarities can be pointed between the concepts of runtime verification and the objectives of passive testing described in the previous section. In general terms, the application of runtime verification techniques can be considered as a form of testing, in particular, since the behavior of the system is being evaluated against some correctness property. The general objectives, however, are slightly different. While testing techniques (and particularly conformance testing) have as objective to provide an evaluation of the system with respect to its requirements, runtime verification in general deals with the technical aspects of evaluation of properties on particular executions and generation of monitors, without necessarily attempting to provide a specific verdict on the system. Nevertheless, this does not exclude the fact that monitors may be designed and used to test conformance properties. One last difference can also be mentioned, particularly in the context of conformance testing described before, which is the fact that techniques in testing treat the system as a black-box, limiting the trace to contain only observable events (inputs/outputs). In contrast, a trace in runtime verification can be any sequence of states of the system.

Chapter 3

The IMS and the Session Initiation Protocol

Contents

3.1 The Session Initiation Protocol

The purpose of the current chapter is to provide an overview on the IMS before going into a study of methodologies for testing IMS services (in Chapter 4). As it will be explained in the next section, the main protocol used for signaling and session establishment in the IMS is the Session Initiation Protocol (SIP) and many entities in the IMS architecture behave partly as SIP entities. Therefore an overview of this protocol, its objectives and mode of operation, is necessary for understanding concepts of the IMS.

3.1.1 Overview

The Session Initiation Protocol (SIP) is an application-layer control (signaling) protocol specified by the IETF (RFC 3261 [Rosenberg 2002]) for creating, modifying and terminating multimedia sessions with one or more participants, independently of the underlying transport. It is the protocol chosen by 3GPP to be the session control protocol for the IMS [3rd Generation Partnership Project (3GPP) 2008], choice that coincided with the IETF work to update SIP from the previous RFC 2543 [Handley 1999] in order to support wireless environments. Although SIP does

not provide services, it provides a flexible and extensible set of primitives which are ideal for the implementation of different services, as it will be seen later in this chapter, when describing the Presence and Push-to-talk Over Cellular services in the IMS.

A typical SIP session is established as follows, where a user (Alice) calls another user (Bob). A diagram of the entities in the communication and the message exchange are provided in 3.1. A more detailed description of this example can be found on the Section 4 of the RFC 3621.

1. Alice uses a SIP client software on her PC, which can act as a User Agent Client (UAC, when sending a request) or User Agent Server (UAS, when receiving a request). Alice calls Bob using his SIP identity, a type of Uniform Resource Identifier (URI).
2. The client, acting as a UAC, generates a SIP `INVITE` message, similar to an HTTP message, containing a request line indicating the method (`INVITE`) the callee identifier (`sip:bob@domainB.org`) and version, followed by a number of headers, as shown in the Figure 3.2. The message also contains a body characterizing the preferred session configuration expressed using the Session Description Protocol (SDP).
3. If the software client does not know the IP address of Bob, it locates a proxy server inside own domain (`domainA.org`), where the message is transmitted.
4. The proxy from the calling domain, sends a `100 Trying` response to the UAC to let it know that it is processing the request.
5. The proxy locates a proxy in the reception domain (`domainB.org`), where it sends the message, first adding its own address to the `Via` header, to keep track of the transmission path.
6. The proxy in the receiving domain, sends a `100 Trying` response to the proxy in the sending domain to let it know that is processing the request.
7. The proxy locates the address of Bob by consulting, for instance, a location server (database mapping SIP identifiers to IP addresses) and transmits the message to that address, again adding its own address in the `Via` header.
8. The client on Bob's end receives the message and returns a `180 Ringing` response, indicating that it is waiting for Bob to answer the call. The response follows the same path that the original `INVITE` request, thanks to the information in the `Via` header.
9. When Bob answers the call, the client software sends a `200 OK` to indicate that the call has been answered, the response message also contains an SDP body, indicating the parameters of the session that Bob is willing to establish.
10. Once the client on Alice's side receives the `OK` response, it immediately sends an `ACK` request to acknowledge the reception of the message, and starts the

media session.

11. When the media session is over, the terminating client sends a `BYE` message, which is replied with a `200 OK` response.

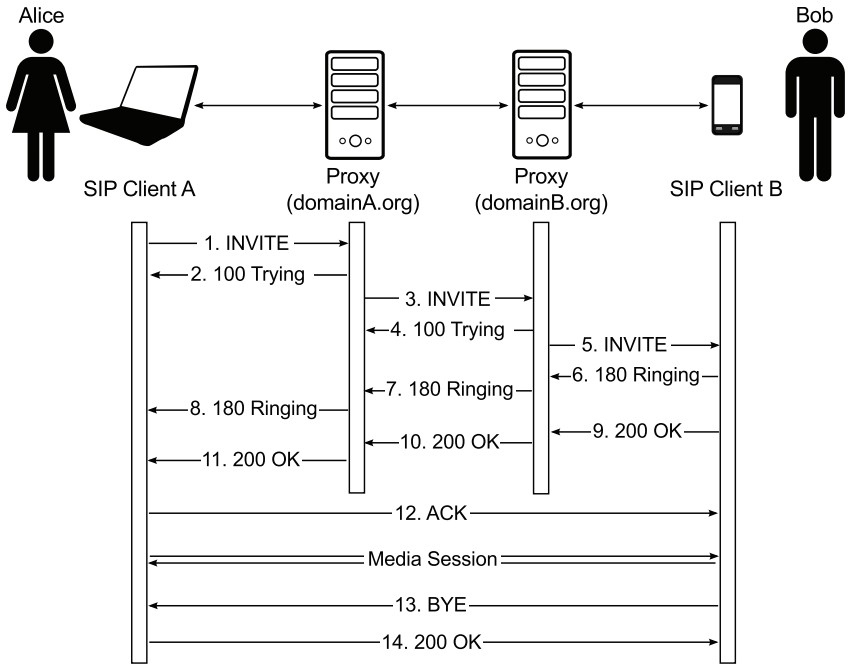


Figure 3.1: SIP entities and message exchange in a typical session establishment.

```
INVITE sip:bob@domainB.org SIP/2.0
Via: SIP/2.0/UDP pcA.domainA.org;branch=z9hG4bK776asdhds
Max-Forwards: 70
To: Bob <sip:bob@domainB.org>
From: Alice <sip:alice@domainA.com>;tag=1928301774
Call-ID: a84b4c76e66710@pcA.domainA.org
CSeq: 10 INVITE
Contact: <sip:alice@pcA.domainA.com>
Content-Type: application/sdp
Content-Length: 142

(SDP body not shown)
```

Figure 3.2: Example of the call initiating `INVITE` message from Alice to Bob

In what follows, we provide a brief description of the behavior of the different entities in the SIP protocol, focusing particularly on the exchanges of messages required for establishing communication, and how the data inside the messages provides the information that allows user agents to decide the course of action on a particular event. Since the focus of the work in this thesis is on passive testing, our main interest is the reflection of the internal behavior of entities in the messages in

the trace.

3.1.2 Entities and Network Elements

Some of the entities and elements that take part in a SIP session are described as follows, as these concepts will be used during the rest of the document.

User Agent or UA is the endpoint in the SIP communication in charge of generating requests and responses, therefore in charge of the communication. A UA can take the role of either an *User Agent Client* (UAC), which is in charge of creating and sending requests, or an *User Agent Server* (UAS) which generates the responses.

Proxy Server An intermediary entity that acts as both a server and a client for the purpose of making requests on behalf of other clients. A proxy server primarily plays the role of routing, which means its job is to ensure that a request is sent to another entity "closer" to the targeted user.

Registrar is a SIP server that receives SIP `REGISTER` requests and stores the information in those requests, i.e. the SIP URI and IP pair, to the location service for its domain so other entities can locate the user later.

Redirect Server A redirect server is a user agent server that generates 3xx (redirect) responses to requests it receives, directing the client to contact an alternate set of URIs.

3.1.3 Message Syntax

As said previously, the syntax for SIP messages is similar to that of HTTP. Each message begins by a start line, called the request line (if the message is a request) or a status line (if a response). The start line is followed by a number of headers and the message body.

The request line is composed by the method of the request, indicating the type of operation requested, the request URI, indicating the user or service being addressed, and the version of SIP used in the message. A short description of the methods defined in RFC 3621 is provided as follows.

- `REGISTER`: used by the UA to indicate its current SIP address and the SIP URI being used as identifier.
- `INVITE`: used to initiate a media session between UAs. It is the most important method for SIP communication.
- `ACK`: used to acknowledge the reception of a message, usually a 2xx response.

- `CANCEL`: used to terminate a previous request, e.g. to hangup the call before having a response.
- `BYE`: used to terminate an ongoing media session.
- `OPTIONS`: used to query a server of its capabilities.

As mentioned earlier, the SIP protocol allows extensions, which introduce new modes of session control by addition of new methods and/or headers. For instance for the Publish service, RFC 3265 [Roach 2002] introduces the `SUBSCRIBE`, `NOTIFY` methods for subscription and notification of events, and RFC 3903 [Niemi 2004] introduces the `PUBLISH` method for providing updated information to a server.

A response's status line, is composed by a status code, a 3-digit integer indicating the outcome of a request, and a reason code, providing a short textual description of the status code intended for a human user. Status codes define six different classes of response.

- `1xx`. Provisional: indicating that the request has been received and the process is being continued.
- `2xx`. Success: indicating that the action was successfully received, understood and accepted.
- `3xx`. Redirection: further action needs to be taken in order to complete the request.
- `4xx`. Client error: the request contains bad syntax or cannot be fulfilled.
- `5xx`. Server error: the server failed to fulfill an apparently valid request.
- `6xx`. Global failure: the request cannot be fulfilled at any server.

SIP headers follow similar grammar rules to HTTP headers. A header line starts by the header name, followed by a colon and the header value, ending in a carriage-return line-feed sequence (CRLF). Groups of headers with the same header name are considered equivalent to one header followed by a comma separated list. The following six header fields are the mandatory minimum for any request formulated by a UAC according to the RFC.

- `To`: specifies the desired logical recipient for the request in the form of a SIP URI or another URI scheme. It is usually composed of the contact display name (optional), as information for the human-user, the identifier of the target, as well as other optional parameters.
- `From`: indicates the logical identity of the user initiating the request. It also contains an URI as the identity and an optional display name. It is used to determine the processing rules to apply to a request (e.g. for call rejection).

- `CSeq`: serves as a way to identify and order transactions (described in the next section). It consists of a sequence number and a method, where the method matches the method from the request line or, if the message is a response, it matches the method of the request being responded.
- `Call-ID`: acts as a unique identifier to group together a series of messages. It identifies uniquely a particular invitation or all registrations of a particular client.
- `Max-Forwards`: serves to limit the number of hops a request can transit on its way to a destination. It is an integer that is decreased at each hop. If the value of the header reaches 0 before reaching its destination, a 483 response is produced.
- `Via`: indicates the transport and addresses of each location where the message has gone through in order to arrive at its destination. Each time a request goes through a hop, the local UAC inserts new address in the `Via` header of the request. That way, the response can use the same path as the request did in the opposite direction.

The message format requires that a blank line is used to indicate the end of the header section in a message, and the beginning of the message content section. Other headers and more detailed description of data inside each header will be provided in the rest of the work, as necessary for examples.

3.1.4 SIP Transactions and Dialogs

In SIP, multiple messages are exchanged during session establishment, for negotiation of media properties and also to maintain peers informed of the status of the setup (trying, ringing, ok, etc.). As seen on Section 3.1.1, the particular sequence of messages, starting with an `INVITE` and finishing with an `200 OK`, allows to effectively establishing a call. Such sequence of messages, starting by a request, followed by a series of provisional responses, and one or more final responses, is called a *transaction* in SIP, and is an essential concept for understanding the behavior of SIP user agents. In the following, we briefly describe the different types of transactions defined by SIP and the relation to the behavior of user agents, the identification of transactional data in messages, as well as the grouping of transaction into dialogs.

In addition to identifying a particular sequence of independent message exchanges, the term transaction also identifies the logical component or layer inside the UA that handles such sequence of messages. This component is called a *client transaction* when it sends the requests (in the UAC), or a *server transaction*, when it sends the responses (in the UAS). Transactions maintain the status of the communication, handle retransmissions (e.g. when the underlying transport does not) and timeouts. The behavior for the client and server transactions are formally specified as state machines in the RFC. It should also be mentioned that in the case of

stateful proxies, two transactions are created, one server transaction to process the requests, and one client transactions to generate the new requests for the next hop in the transmission.

Two types of transactions are defined by the SIP specification: INVITE and non-INVITE, each with client and server sides. INVITE transactions handle the three-way handshake (`INVITE` – `OK` – `ACK`) described on the example in Section 3.1.1, and include the generation of the `ACK` message (on the client side), and retransmissions of the `OK` (on the server side) while waiting reception of `ACK`. Non-INVITE transactions do not make use of `ACK`, they only handle simple request-response interactions, as the `BYE` – `200 OK` transaction in the previous example.

The INVITE client transaction, as illustrated by Figure 3.3 is created by the UAC[1] and the `INVITE` message created by the client is provided as an initial input to the transaction, leaving the machine in the 'Calling' state and prompting the machine to send the message (described with the action in the starting transition, denoted by '**A**'). In the figure, the timer A controls retransmissions when underlying transport is unreliable, timer B controls the transaction timeout and timer D defines the maximum time that the state machine can be in the 'Completed' state. The actions taken by the transaction upon different events are denoted by '**E**' in each transaction text and are described in detail on Section 17.1.1.2 of the RFC, therefore we will not further explain here. Similar descriptions are also provided in the specification for INVITE server transaction and non-INVITE server and client transactions. Most behavior in SIP communication can be described by these four transactions.

When a UA receives a request or a response, it has to determine which transaction the response belongs to, so the processing of the message can take place (or continue) in the appropriate state machine. The UA uses the following rules from the data in the headers to match a transaction

1. The `branch` parameter in the top `Via` header (added in the last hop) in the request or response must match the one from the request that created the transaction.

2. The method in the `CSeq` header of a response (or request, except for `ACK`) must match the method of the request that created the transaction.

3. For requests, the value of the `sent-by` parameter in the top `Via` header must be equal to the one of the request that created the transaction.

These rules will be useful when we get into passive testing for SIP services, in Chapter 4.

In addition to the grouping established by a transaction in SIP, a higher level grouping also exists in the concept of a *dialog*. A dialog represents a peer-to-peer

[1] Specifically transactions are created by the Transaction User or TU, the layer right above the transaction layer which deals with creation and management of transactions.

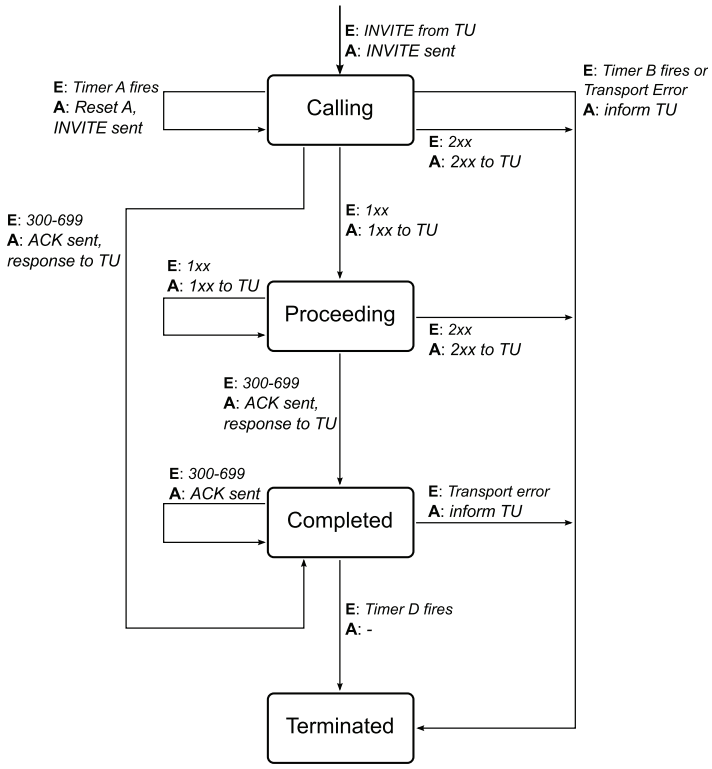


Figure 3.3: INVITE client transaction.

SIP relationship between two user agents that persists for some time. The dialog facilitates sequencing of messages between the user agents and proper routing of requests between both of them. In the example from Section 3.1.1, the INVITE transaction establishes a dialog, and the information from the `Record-Route`, `Route` and `Contact` headers is used in the `BYE` – `200 OK` transaction, in order to transmit end-to-end between the UAs. Other methods can also create dialogs (e.g. `SUBSCRIBE`), however only the dialog created by `INVITE` is described in RFC 3621. A dialog is recognized in the messages by the `Call-ID` header and the local and remote `tag` parameters in the `From` and `To` headers, where the local tag is placed on the `From` header in the request, and moved to the `To` header in the response. The Figure 3.4 illustrates the relation between dialog and transaction in a call.

3.2 Overview of the IMS

The IP Multimedia Subsystem [Camarillo 2005, Ahson 2008] (IMS) is a standardized framework for delivering IP multimedia services to users in mobility. It was originally intended to deliver Internet services over GPRS connectivity. This vision was extended by 3GPP, 3GPP2 and TISPAN standardization bodies to support more access networks, such as Wireless LAN, CDMA2000 and fixed access networks. The IMS aims at facilitating the access to voice or multimedia services in

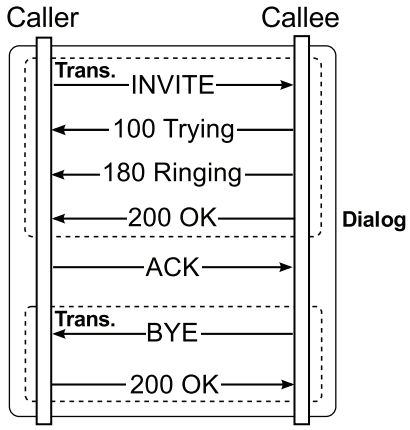


Figure 3.4: Relation between transactions and dialogs in SIP session establishment.

an access independent way, in order to develop the fixed-mobile convergence. To ease the integration with the Internet world, the IMS heavily makes use of IETF standards (e.g. SIP).

In the current section we provide an overview of the architecture of the IMS, briefly describing the different entities that interact during a session establishment in the IMS, as well as the protocol used for communication. Then we provide a description of two of the services that will be used during the rest of the work, the Presence service and the Push-to-talk Over Cellular (PoC).

3.2.1 Core architecture

The core of the IMS network consists on the Call Session Control Functions (CSCF), that redirect requests depending on the type of service, the Home Subscriber Server (HSS), a database for the provisioning of users, and the Application Server (AS), where the different services run and interoperate. Most communication with the core network and between the services is done using the Session Initiation Protocol [Rosenberg 2002]. Figure 3.5 shows the core functions of the IMS framework and the protocols used for communication between the different entities.

3.2.1.1 Call/Session Control Functions

The call/session control functions (CSCFs) act as SIP servers and are the nodes that process the SIP signaling in the IMS. Depending on the function and the type of session, they handle authentication, compression, encryption and routing of messages. They can be located either in the visited network (when roaming) or the home network.

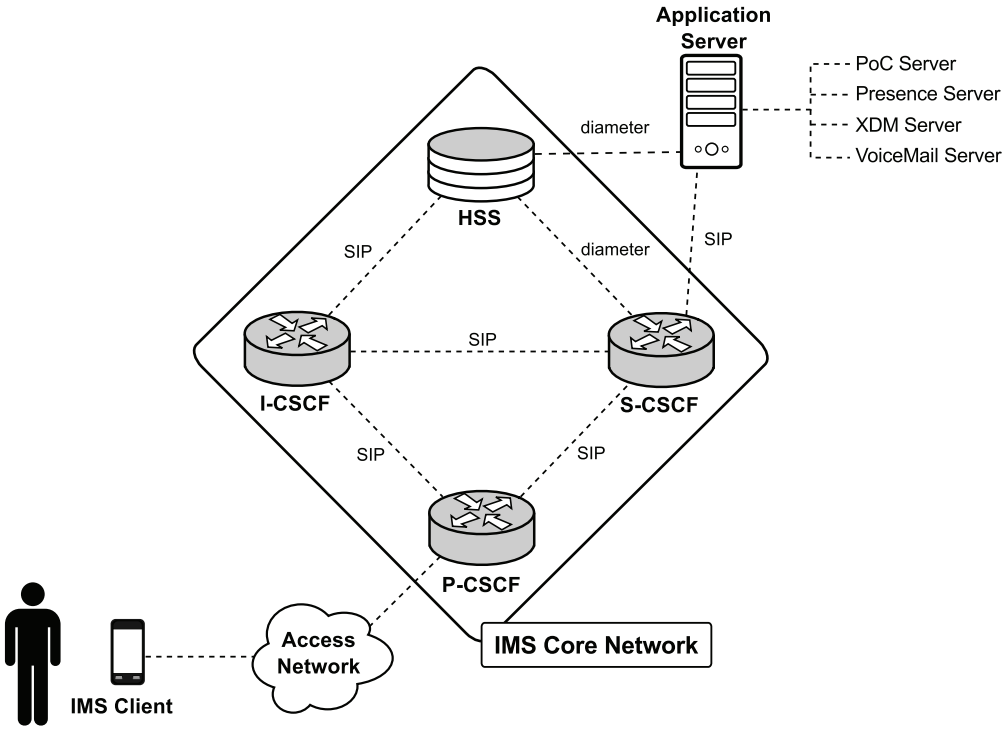


Figure 3.5: Core functions of the IMS framework.

3.2.1.2 Home Subscriber Server

The Home Subscriber Server (HSS) is the central repository/database containing all user related information, including between others, location information, authorization and authentication information, and the S-CSCF allocated to the user. A network may contain more than one HSS, but all the information of a particular user is stored in a single HSS. When this is the case, a Subscription Locator Function (SLF) is used to map users to particular HSS.

3.2.1.3 Application Server

The Application Server (AS) is the SIP entity where services are hosted and executed. Depending on the service the AS can act as a SIP proxy, SIP UA (User Agent) or SIP B2B2UA (Back-to-Back User Agent). The AS interfaces the S-CSCF and the I-CSCF using SIP and the HSS using Diameter. It can be located either in the home network or in an external third-party network to which the operator maintains a service agreement. Only ASs located in the home network can interface the HSS.

3.2.2 IMS Services

One of the main objectives of the IMS is to provide an architecture for deployment and integration of multimedia services, in the following we will describe in general terms two of these services: Presence and Push-to-talk Over Cellular.

3.2.2.1 Presence

The Presence service is a system to disseminate presence information, allowing a user of the service be informed of another contact's connection status (online, offline) as well as their availability status (idle, busy, in a meeting, etc.). It also allows users to give details of their communication capabilities (e.g., whether they have audio, video, instant messaging, etc.).

The Presence specification (RFC 3856 [Rosenberg 2004]) proposes the usage of SIP as a presence protocol. As such, it assigns user agents for performing the tasks of the different roles in the presence information exchange, roles defined in accordance with the model for presence defined in RFC 2778 [Day 2000]. This is illustrated in the Figure 3.6 and described in the following.

- The user (Alice) sharing his presence information is called the *presentity*, who performs the task by means of one or more *Presence User Agents* (PUAs).
- The updated information is sent by the PUA to a *Presence Agent* (PA), where the general picture of the user's presence is kept. A PA can be part of a *Presence Server* (PS) which is an entity that can act either as a PA or as a SIP proxy for subscription requests.
- Users interested in Alice's presence status, called *watchers*, can subscribe to the PA to be notified of updates to Alice's status.

The watchers subscribe to the PA by using the pair of messages `SUBSCRIBE`/`NOTIFY`, defined in the RFC 3265 [Roach 2002], where the utilization of SIP for event notification is specified. Between other things, the `Event` header is introduced to indicate the type of event to which the subscription/notification is made. In the case of presence, the value of this field is the string "`presence`". The actual presence information is included in the body of the message, in a format depending on the application, which for the IMS is an extension of PIDF (Presence Information Data Format [Sugano 2004]) defined by the Open Mobile Alliance. For updating information to the PA, the presence service uses the SIP `PUBLISH` request. As this request can be used in general for publishing any type of information, then "`presence`" must be used as value for the `Event` header, among other conditions.

For the IMS the main specification for the Presence service is provided in the OMA Presence SIMPLE specification [Open Mobile Alliance 2010].

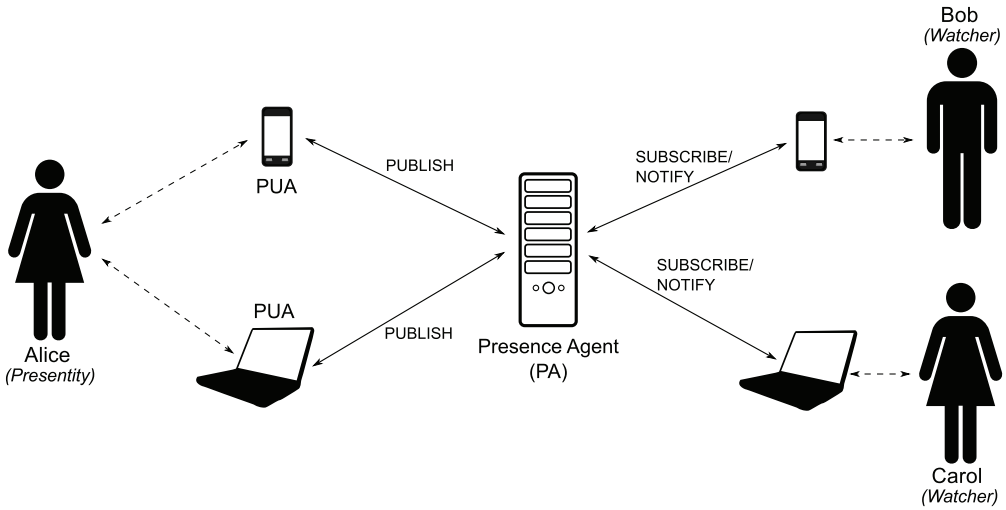


Figure 3.6: SIP entities in the Presence service

3.2.2.2 Push-to-talk Over Cellular

The IMS Push over Cellular (PoC) service, standardized by the OMA [Open Mobile Alliance 2006], is also known as Push-to-Talk, Push-to-View or Push-to-Share, depending on the main media type of the communication. It enables multiple IMS users to connect with each other in a single communication session, where any authorized user may talk simultaneously to every other participants. It is a walkie-talkie communication paradigm (half-duplex), meaning that only one user can speak at a time.

For the service two planes are defined: the control plane, which deals with session setup using SIP, and the user plane, dealing with the assignation of talking rights, described by the specification as media bursts (or talk bursts in earlier versions of the documents), and specified in the protocol MBCP (Media Burst Control Protocol). From an architecture point of view, two roles or functions are specified on the server side: the controlling function, providing centralized session handling and media distribution, and the participating function, providing session handling and media distribution on the side of the client (the home network). The communication between two clients is illustrated in Figure 3.7.

For signaling, several IETF standards are used as extensions or complements to SIP. The RFC 4354 [Garcia-Martin 2006] defines the '`poc-settings`' event, for publication and notification of configuration changes in the PoC client (using the SIP notification extension described for presence [Roach 2002]), as well as the XML schema for distribution of the settings. RFC 5363 [Camarillo 2008] specifies the framework for URI-list services, which permits to clients to perform operations that involve several users (as an invitation to a conference call), by specifying them in URI-lists, an XML document included in the body of the SIP message (defined also in RFC 5363). Finally, the headers `P-Answer-State`, `Answer-Mode` and `Priv-Answer-Mode`

are defined in RFCs 4964 and 5373.

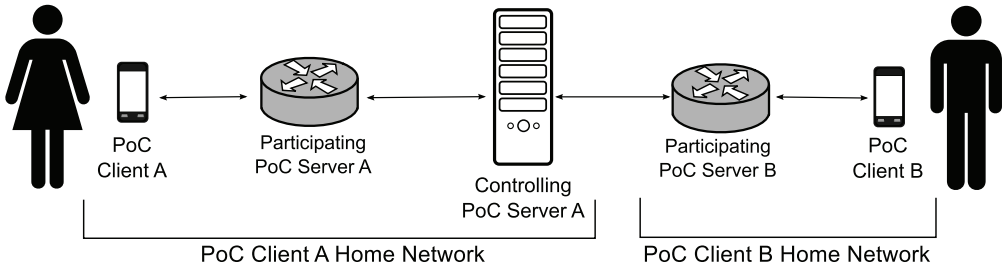


Figure 3.7: Controlling and Participating functions in a PoC session

Different types of sessions are possible in PoC. A *One-to-One* session allows communication between two users. An *Ad-hoc PoC Group* session is established when the inviting user choses a group of contacts from the address book and invites them to a multi-party PoC session. A *Pre-Arranged PoC Group* is similar to an ad-hoc group however group have been selected in advance. A *Chat PoC Group* is a multi-party session where a user can join or leave the communication as desired.

The figure 3.8 shows the session setup for an ad-hoc group session with confirmation (the invitees respond with `180 Ringing` and wait for the user to accept before sending the `200 OK` response), and illustrates the number of message exchanges required for setting up some sessions. The figure does not include the exchange between entities of the SIP Core (the CSCF servers) and we also omitted the `100 Trying` messages generated each time a server receives an `INVITE` request, and the `ACK` request sent for acknowledgement by inviting user after the `200 OK` has been received. As shown in the figure, the `INVITE` from the inviting user contains an `URI-list` with the list of invitees. When the controlling PoC server receives the message, it generates an invitation for every client in the list. In terms of SIP entities, this means that the PoC server creates a new client transaction for every client in the `URI-list`,

Once the session has been established, the user plane takes control of the communication. As mentioned earlier, the floor control is provided by the MBCP protocol, an extension of RTCP (Real Time Control Protocol). Once the `200 OK` has been received, a `MBCP Media Burst Granted` message is sent to the client that initiated the call in order to grant him the floor. Other clients can request authorization to speak by sending a `MBCP Media Burst Request` to the server. Other messages are also used to acknowledge a message, to deny a request or to inform the clients that the floor is free. The final media communication is done using RTP (Real Transmission Protocol) with the media configuration negotiated during session setup.

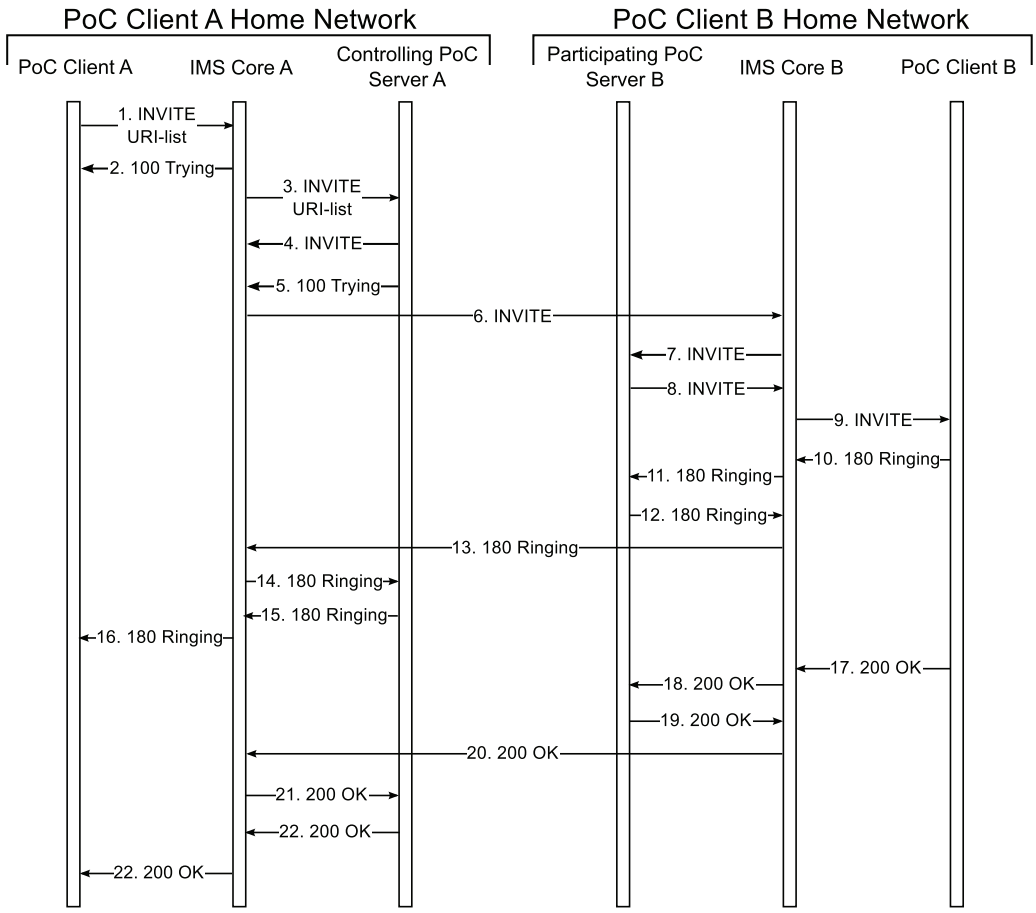


Figure 3.8: Message exchange for an Ad-hoc Group Session with confirmation.

Chapter 4

Testing for IMS Services

Contents

4.1 Introduction

As the number of inter-operated IMS implementations grows, the number of complex IMS applications and services is also increasing [Tsagkaropoulos 2007]. Moreover, the fact that people rely on computers in practically every aspects of their lives (e.g. in ATMs, smartphones, netbooks, cars, etc.) makes the cost of unreliable design higher [Hoffman 2008]. Even given the importance of IMS applications for the commercial success of the framework, and the important requirements of inter-operability between platforms established by the IMS (IMS core, application server, client applications across multiple networks), few works have been done with regards to conformance testing of IMS services. There are different reasons that could explain this. One of them is the necessity for the industrials to quickly provide new services to their clients in this new competitive race which is the fixed-mobile convergence. Another reason is the specific black-box nature of IMS implementations. Due to many issues, but mainly due to commercial aspects, the access to

available interfaces (with little available knowledge about them) to *actively* test the implementations is usually limited.

Some works can be mentioned related to testing for the IMS. Modeling of SIP services has been done by the authors of [Chan 2003b, Chan 2003a] showing that SDL is a suitable language for modeling SIP services and introduce an approach to reduce services feature interactions. Concerning IMS testing, many works have been done for IMS testbeds. A couple of examples can be cited, for instance, in [Blum 2007], aimed at studying the impact of new access networks, or [Panwar 2007] to test IMS clients by simulating server functionality. Most works make use of the Open IMS Playground of Fraunhofer Fokus[1]. More recently, some works have been introduced regarding testing of IMS services using TTCN-3 [Bormann 2009]. Finally tests definitions can be found for testing different IMS core requirements in the TTCN-3 website[2].

As seen in Chapter 3, IMS services integrate multiple protocols and protocol extensions in order to provide features of the service. For instance in the case of the Push-to-talk Over Cellular (PoC) service, SIP is used for session establishment, PoC settings exchanges is defined using RFC 4354 [Garcia-Martin 2006], conference status notifications are used according to the definitions in RFC 4575 [Rosenberg 2006] (an extension of RFC 3265 [Roach 2002]). Conformance testing of IMS services must consider then two separate aspects:

1. That each protocol and protocol extension implemented by the service is conformant to its specification. For instance, it should be validated that, when the PoC server behaves as a SIP UAC for contacting invitees of a conference, the observed behavior of the server is consistent with the INVITE client transaction, as specified by RFC 3261.

2. That the integration of the different protocol extensions conforms to the requirements of the service. In the PoC, for example, if a client subscribes to conference status (RFC 4575), then upon change of subscription state (e.g. another client joined the conference), the server must send a `NOTIFY` message to the subscribed user [Open Mobile Alliance 2009].

We presented an approach to deal with some of these testing challenges [Lalanne 2009b, Lalanne 2009a], through an industrial case study performed as part of our participation in the ExoTICus[3] project. There, an invariant-based passive testing methodology was used for testing the behavior of server and client in a specific session scenario of the PoC service, the Ad-hoc Group session establishment. The presented approach consisted of the following steps: first the properties to be tested are defined as invariants, properties are defined from the information provided

[1] `http://www.openimscore.org/`
[2] `http://www.ttcn-3.org/PublicTTCN3TestSuites.htm`
[3] `http://www.systematic-paris-region.org/en/projets/exoticus`

in the description of the session and example of Message Sequence Charts (MSCs) provided by the PoC control plane specification [Open Mobile Alliance 2009, Sections 7.2.1.2 and F.5.1] Secondly, these requirements are verified on a formal specification for the PoC, using the SDL specification language. Finally, these properties are evaluated in execution traces of the service, obtained from the IMS implementation of an industrial partner, Alcatel-Lucent.

In this chapter, a partial description of that work is provided, particularly focusing on the property definition. Although the work described is a first approach, it served to raise interesting question that motivated the research presented in this thesis, that will be detailed in Chapter 5, particularly in relation to the importance of data in invariant-based approaches. These motivational ideas are described in 4.4.2.

4.2 An Invariant-based passive testing approach

4.2.1 Obligation Invariants

Given the constraints of the platform, obligation invariants [Bayse 2005] (Definition 2.1.5) were used as part of the testing methodology proposed for IMS services. Since SIP communication is dependent of the data inside the message headers, the concept of invariant used is closer to that of *backwards invariants* in [Ladani 2005]. The main difference is that input/output symbols in invariants from the latter approach can optionally contain parameters, as input/outputs in an EFSM (Definition 2.1.3). Let us define invariants for an EFSM and briefly recall some of the ideas of this approach.

According to Definition 2.1.5, an invariant I is described by $I ::= i/\overline{O} \mid *, I \mid i/o, I$, where $i \in \mathcal{I} \cup \{?\}$ is an input, $o \in \mathcal{O} \cup \{?\}$ is an output and $\overline{O} \subseteq \mathcal{O}$ is a set of obligation outputs. Intuitively, a sequence such as $\{i_1/o_1, ..., i_{n-1}/o_{n-1}, i_n/o_n\}$ is an obligation invariant for M if each time i_n/o_n is observed, then the trace $i_1/o_1, ..., i_{n-1}/o_{n-1}$ happens before. An invariant that expresses a property, such as "if y happens then we must have that x has happened before", is an obligation invariant. They may be used to express properties where the occurrence of an event must be necessarily preceded by a sequence of events.

In addition to sequences of input and output symbols, the wild-card characters '?' and '$*$' are allowed, where '?' represents any single symbol and '$*$' represents any sequence of symbols.

In this work, invariants are checked both on the specification and on the IUT, allowing to determine whether the property is correct according to the formal model and that the IUT behavior is as defined by the standard. An adaptation of a classical string pattern matching strategy [Knuth 1977] is performed to match the trace and the invariants. The complexity of the resulting algorithm is in the worst case in $O(m \cdot n)$ (n being the invariant length and m the trace length), in particular because we must check all occurrences of the pattern in the trace making it difficult

to optimize the complexity to $O(m)$. Nevertheless, since the invariant length is often (not to say "always") much smaller than the extracted trace, the complexity is almost linear with respect to the trace length.

4.2.2 The Conformance Passive Testing approach

In the current approach to conformance testing, we attempt to test the correctness of an implementation through the evaluation of a set of invariants (or properties) and a set of captured traces (extracted from the running implementation's points of observation). Four steps are followed during the conformance passive testing procedure.

Step 1 Definition of properties. The relevant protocol properties to be tested are provided from the standard documents or by protocol experts.

Step 2 Formalization of properties as invariants. Properties have to be formulated by means of obligation invariants that express a requirement regarding the local entity under observation. Moreover, the properties are formally verified on the formal specification ensuring that they are correct with respect to the requirements.

Step 3 Extraction of execution traces. In order to obtain such traces, different PO are set up by means of a network sniffer installed on one of the nodes. The captured traces are in XML format.

Step 4 Test of the invariants on the traces. The traces are processed in order to obtain information concerning particular events as well as relevant data (e.g. token owner, source and destination address, origin of data to initialize a variable, etc.). During this processing, the test of the expected properties is performed and a verdict is emitted (Pass, Fail or Inconclusive). An inconclusive verdict may be obtained if the trace does not contain enough information to allow a Pass or a Fail verdict.

4.2.3 The PoC Service

The Push-to-talk Over Cellular service (Section 3.2.2.2) is provided by a PoC server. Session establishment is done using SIP, while the media communication is transmitted via RTP directly between terminals. Floor control is provided via the Talk Burst Control Protocol[4] (TBCP), an extension of RTCP. In the user plane, a *PoC token* is assigned to the client with the floor. The floor is requested/released by the different clients of the session through TBCP `REQUEST` and `RELEASE` messages, and information advertising the status of the token is provided through TBCP `GRANTED`, `DENY`, `IDLE` and `REVOKE` messages. Once the token is granted to a PoC participant,

[4]TBCP becomes Media Burst Control Protocol (MBCP) from version 2.0 of the PoC service

this latter has the opportunity to send media packets while all other participants to the session can only receive them. The PoC token is automatically released after a predefined duration.

For simplicity, a single scenario from the PoC requirements was chosen both for modeling and definition of the properties: The Ad-hoc Group session, or as referred in the PoC requirements document [Open Mobile Alliance 2006], the Selective Dynamic Group Call. This feature provides a way for a service user to quickly set up a one-to-one group call in a dynamic way, without need to specifically define a group in the provisioning server. The normal flow of the session is the following.

- The initiating user selects one or more persons from a contact list user interface, that he would like to establish a call with. The user can use the contact list's presence information to decide who to include in the call.
- After finishing selecting the group members, the user presses and holds the POC button/key, initiating a call with the network, the user will receive a notification to talk (usually by an audio tone) when the first invitee joins the call, indicating that the user can begin to talk.
- Each one of the invitees will receive a notification indicating an incoming group call, and will have the chance to take or reject the call. If they choose to take the call, then they will receive a wait-to-talk indication.
- The session remains active as long as two or more members are engaged in the call.

The general PoC functionality is divided into two planes. The control plane and the user plane. The control plane deals with the media communication and floor control, or talk burst control. The control plane deals with session establishment and control using SIP. Nevertheless, since our approach is black-box, we can only distinguish the two planes through the messages they produce. The Message Sequence Chart (MSC) in Figure 4.1 shows a detailed view of the Ad-Hoc session establishment, also described in the previous chapter (Figure 3.8). After the session establishment, through the `INVITE` – `OK` messages, an `ACK` message is sent from the original client to the PoC server, which prompts the user plane to grant the floor to the original user (through `TB_Granted`) and deny the floor to the rest of the participants (through `TB_Taken`), before sending the corresponding `ACK` messages to the rest of the participants.

4.3 Experiments

4.3.1 Experimental architecture

The Exoticus IMS core network, provided by Alcatel-Lucent, is a full-featured IMS infrastructure designed to support 10,000 users. The mandatory IMS services are

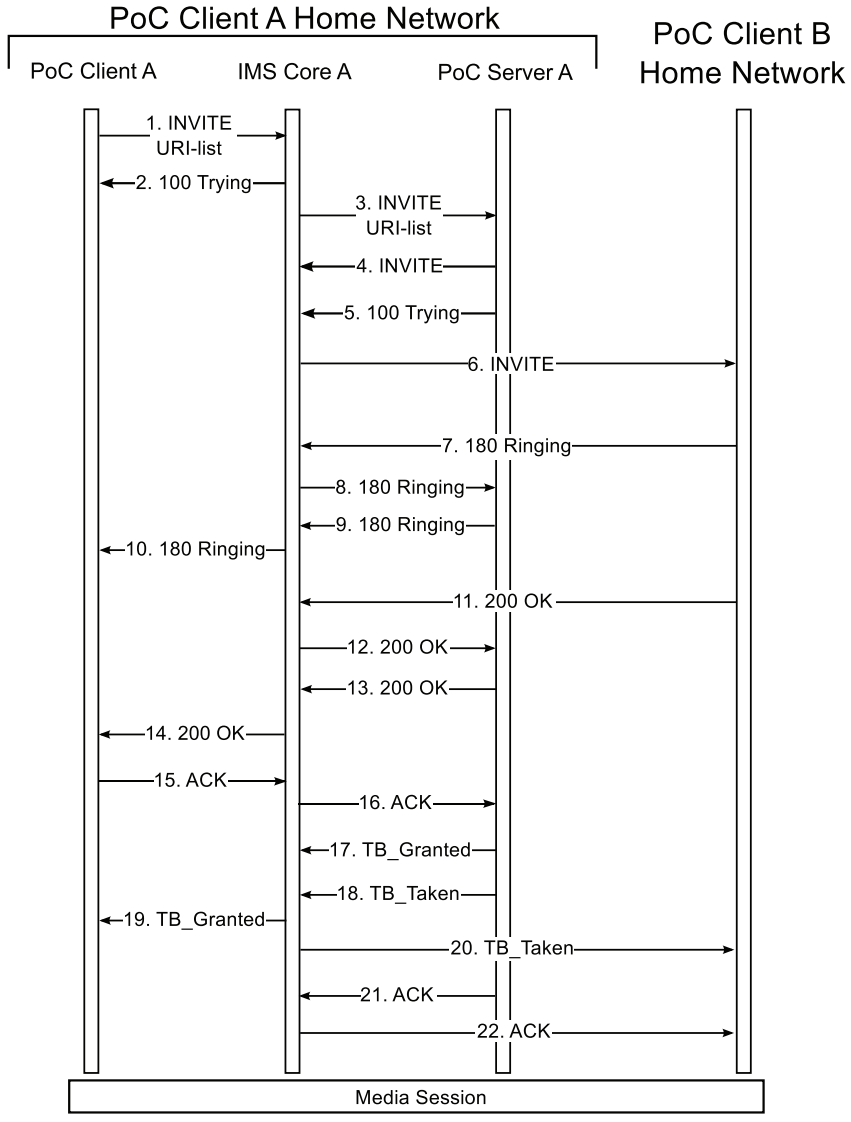


Figure 4.1: Message exchange for an Ad-hoc Group Session.

available from Alcatel-Lucent portfolio, running on the A5400 IMS Application Server:

- The XDM server is the OMA 1.1 standardized network address book manager. It provides one of the most interesting IMS features, which is the centralization of the users' address books. Whatever terminal the user will use to access the IMS, an up-to-date address book will be available. It is stored on an XML documents management server, which can be accessed using the XCAP protocol.
- The second IMS key service is the Presence Server that delivers the OMA 1.1 standardized service for users' presence and status management.
- The voice communication service is not based on circuit switching but on IP packets exchange, and depending on the IMS clients supported audio/video support the communications can be enriched with video.
- The voice mail server provides IMS users with the must have voice and video messaging system.
- The PoC server implements the OMA 1.1 PoC service.

The Figure 4.2 depicts the architecture. The *Point of Observation*, indicated in the figure, illustrates the interface for capture of the PoC service traces.

4.3.2 Invariants

From the OMA PoC requirements, six obligation properties have been provided, to test the behavior of the PoC server as well as some of the behavior of the clients. Each one of these properties describes a step in the session establishment for a PoC Ad-hoc session and allows to ensure that the sequences of messages defined by the protocol appears in the trace. Although the initialization sequence is clearly defined by the PoC, the exchange of messages is asynchronous, and during this process, the server can also exchange SIP messages with other IMS applications and peers of the session. The invariants are described in the following.

4.3.2.1 Invariant 1

The first invariant evaluates the case where a particular user initiates an Ad-Hoc call. The originating user sends an `INVITE` request to the PoC server, identifying it by the conference URI, a constant identifier defined in the client. The `INVITE` message must contain a list indicating the invitees to the session (identified by their URIs), in agreement with the Ad-Hoc session requirements. For each of the users in the list, the PoC server creates a new `INVITE` request, using the originating user's

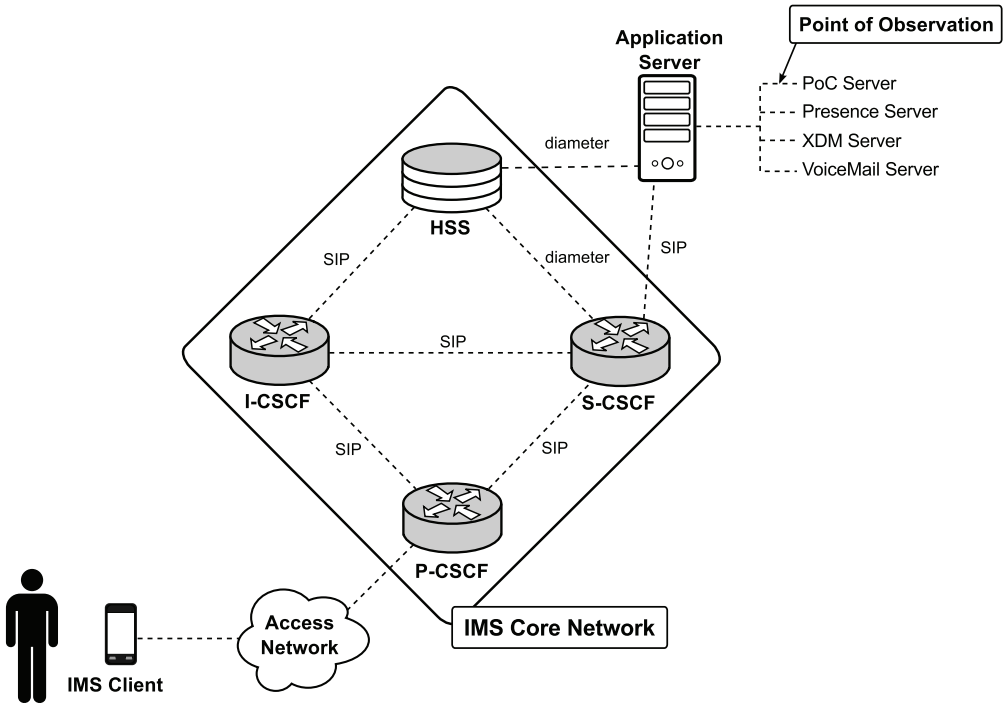


Figure 4.2: IMS testing architecture.

URI as value for the `From` header. The property is defined as the following obligation invariant

$$INVITE(CSeq_0, User_0, \text{`}\texttt{ConferenceURI}\text{'}, CallID_0, \{User_1\})/?, *,$$
$$?/INVITE(CSeq_1, User_0, User_1, CallID_1, \emptyset)$$

where the input symbol is represented by $INVITE(CSeq, From, To, CallId, Invitees)$ where $CSeq$, $From$, To, $CallID$, $Invitees$ respectively represent the `CSeq`, `From`, `To`, `Call-ID` and list of URIs for the invitees. Although it is not specifically denoted in the formula, the last output corresponds to the obligation part of the invariant.

It should be mentioned also that the last type of variable, representing a list of elements, is not technically supported by the definition of invariant given previously. Nevertheless it was supported by the tool used for the experiments. The property is illustrated by the MSC in Figure 4.3.

4.3.2.2 Invariants 2 and 3

These invariants illustrate another step in the initialization sequence. After a SIP client receives an `INVITE` request, it replies with a `Ringing` response, to indicate that the message was received and that it is waiting for confirmation of the user to accept the call. In the case of the Ad-hoc session, reception of `Ringing` response prompts the sending of an equivalent response for the original `INVITE`, to inform the

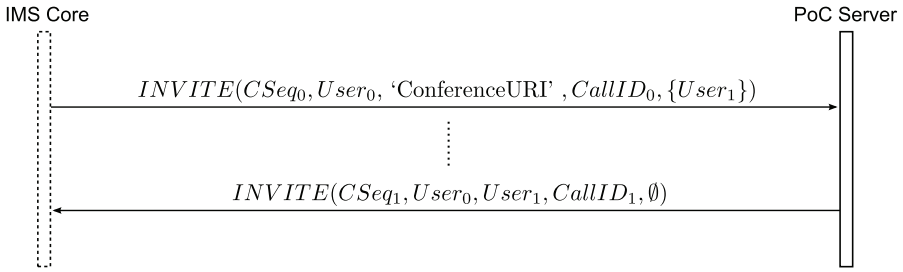


Figure 4.3: MSC of Invariant 1.

client that at least one of the recipients has received the invitation. Property 2 is designed to test the behavior of the clients, and although the traces were gathered at the server, this can be achieved by reversing inputs and outputs in the trace. The property is defined as follows.

$$INVITE(CSeq_1, User_0, User_1, CallID_1, \emptyset)/?, *,$$
$$?/Ringing(CSeq_1, User_0, User_1, CallID_1)$$

Indicating that if a `Ringing` message should not be sent without a corresponding `INVITE` being previously received. Property 3 is defined with the invariant

$$INVITE(CSeq_0, User_0, \text{`}\texttt{ConferenceURI}\text{'}, CallID_0, \{User_1\})/?, *,$$
$$?/Ringing(CSeq_0, User_0, \text{`}\texttt{ConferenceURI}\text{'}, CallID_0)$$

and indicates that if a `Ringing` message is sent, a corresponding `INVITE` message must have been received previously. These properties are illustrated in Figure 4.4.

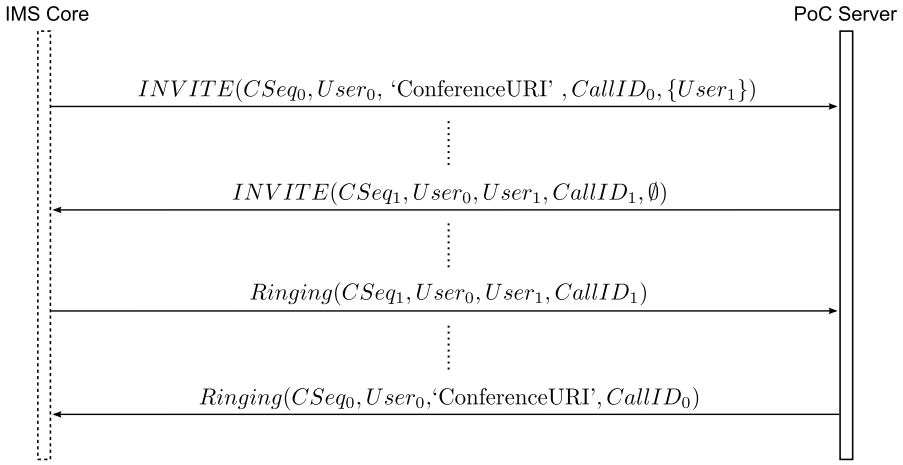


Figure 4.4: MSC of Invariant 2 and 3.

4.3.2.3 Invariant 4, 5

These invariants illustrate the acceptance steps during session initialization. When the PoC client for the invited user accepts the call, an `OK` message is sent to the

PoC server, informing that the client is waiting to listen, which in turn prompts the server to send an `OK` response for the originating `INVITE` request, to inform the caller that at least one invitee has accepted the call. The sequence of messages evaluated by these invariants is shown in Figure 4.5. Invariant 4 is defined to test the behavior of the clients as

$$INVITE(CSeq_1, User_0, User_1, CallID_1, \emptyset)/?, *,$$
$$?/OK(CSeq_1, User_0, User_1, CallID_1)$$

and invariant 5 is defined with the following formula

$$INVITE(CSeq_0, User_0, \text{`}\texttt{ConferenceURI}\text{'}, CallID_0, \{User_1\})/?, *,$$
$$?/OK(CSeq_0, User_0, \text{`}\texttt{ConferenceURI}\text{'}, CallID_0)$$

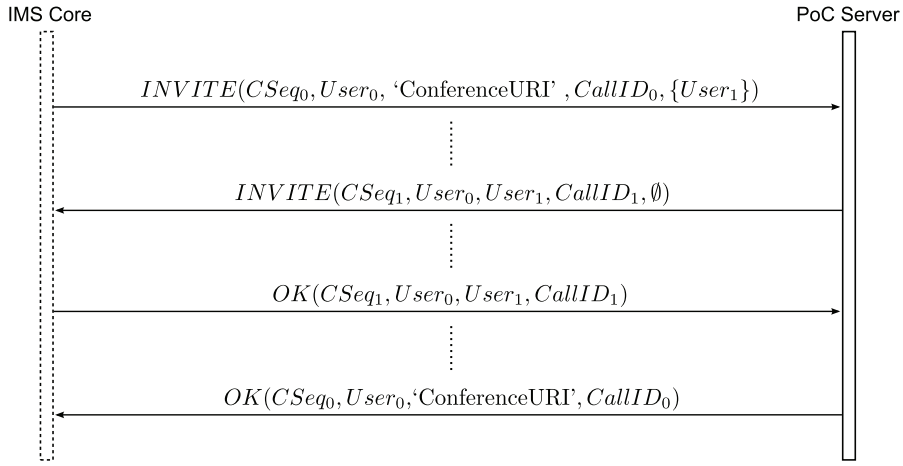


Figure 4.5: MSC of Invariants 4 and 5.

4.3.2.4 Invariant 6

This invariant evaluates the acknowledgement of an OK message by a PoC client. An `ACK` message indicates the PoC server that the `OK` message was successfully received and that the client is ready to initiate the media session. It prompts the PoC server to move to give control to the user plane and grant the floor to the initiating client. This invariant validates the behavior of the client, requiring that before an `ACK` message is received by the server, an `OK` message must have been sent. This property is defined in the following and illustrated by Figure 4.6

$$OK(CSeq_0, User_0, \text{`}\texttt{ConferenceURI}\text{'}, CallID_0)/?, *,$$
$$?/ACK(CSeq_2, User_0, \text{`}\texttt{ConferenceURI}\text{'}, CallID_0)$$

Notice that, in principle, there is no relation between the `CSeq` fields in the property between the `OK` and `ACK` messages. Although the messages are required to belong to the same dialog, as defined by the `Call-ID` header, there is no other guarantee that

the messages are actually related. Nevertheless, given the conditions of test and trace capture, this property is sufficient to evaluate the requirement. A strongest version of the same property can be defined by

$$?/INVITE(<\text{`INVITE'}, Seq_0>, User_0, \text{`ConferenceURI'}, CallID_0, \{User_1\}), *$$
$$OK(<\text{`INVITE'}, Seq_0>, User_0, \text{`ConferenceURI'}, CallID_0)/?, *,$$
$$?/ACK(<\text{`ACK'}, Seq_0>, User_0, \text{`ConferenceURI'}, CallID_0)$$

where the $CSeq$ part of the $INVITE$ event is described by the pair $< Method, Seq >$, where the $Method$ represents the part of the `CSeq` header containing the method, and the Seq contains the sequence part of the header. This also serves to illustrate how the effectiveness of the property relates to number of data variables that need to be compared.

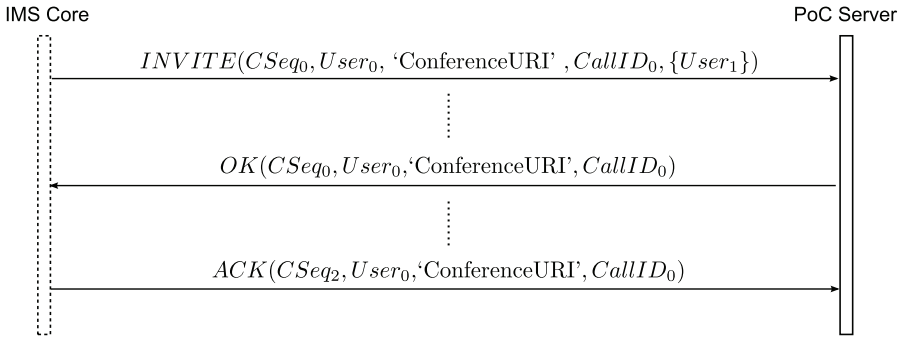


Figure 4.6: MSC of Invariant 6.

4.3.3 Testing tool

The testing tool used, called TestInv and developed by the company Montimage[5], allows automated analysis of the captured traces to determine if the given invariants are satisfied or not. The tool takes as input the information on the protocols under observation, the traces and the invariants for evaluation. Invariants for the tool, in addition to defining the sequences of events that need to be observed, can also specify maximum time limits (*timeouts*) between events. The verdict obtained for an invariant can be either PASSED, FAILED or INCONCLUSIVE meaning respectively that: all events where satisfied; at least one event was not satisfied in the trace in the time delimited by the *timeout*; or, it is not possible to give a verdict because there is not sufficient information in the trace. The Figure 4.7, shows a high level description of the tool as it was defined at the time of the work.

4.3.4 Results

The defined invariants were defined using the XML syntax for the tool TestInv, specifying the types of packets to observe and the events to evaluate. This file,

[5] `http://www.montimage.com`

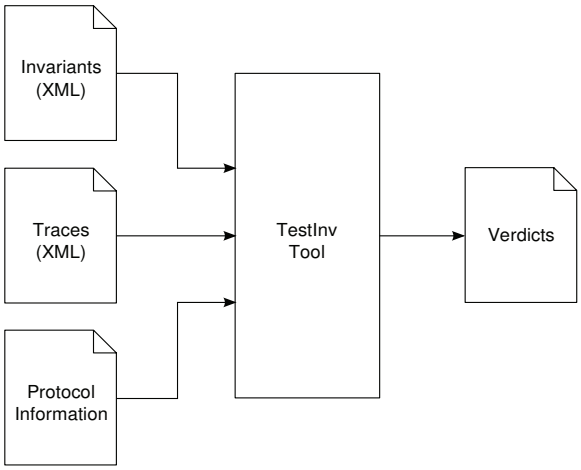


Figure 4.7: The TestInv Tool

along with the trace files and protocol information were provided as input for the tool. The definition of invariants and interpretation of the results is not difficult for someone with knowledge of the protocol requirements. Applying the tool to the traces was very fast (less than 1 sec.) since the traces did not need to be very long for the purpose of testing the PoC service. Some calibration is necessary to find a convenient timeout value to use but this did not present many problems.

The verdicts obtained were PASSED for the majority of the invariants previously described. This means that for each invariant, meaning that in most of the cases, the *preamble* of the invariant was found for each time that the *obligation* part of the invariant was observed. For instance for the Invariant 1, after finding an obligation packet

```
timestamp = Nov 17, 2008 14:30:20.054048000
sip.Request-Line = INVITE sip:fabrice@mediacom.net SIP/2.0
sip.to.addr = sip:fabrice@mediacom.net
sip.from.addr = sip:ronan@mediacom.net
sip.CSeq = 1000 INVITE
sip.Call-ID = 001cc4600e68-29206744931200764419
```

the tool reports finding the preamble packet

```
timestamp = Nov 17, 2008 14:30:19.939698000
sip.Request-Line = INVITE sip:Conference-Factory@mediacom.net SIP/2.0
sip.to.addr = sip:Conference-Factory@mediacom.net
sip.from.addr = sip:ronan@mediacom.net
sip.CSeq = 20 INVITE
sip.Call-ID = 3040056629@172.25.70.116
```

Nevertheless, several FAILED results were also found for the invariants, particularly for property 4. Upon inspection of the traces, these results were determined

to be *false positive* results, i.e. behavior incorrectly detected as a fault during the trace observation. A two part explanation can be provided: *1)* valid behavior from the PoC server, not indicated in the MSC used for the specified scenario, was not considered during property definition, therefore leading to *2)* an invariant not restrictive enough, in terms of data parameters, to correctly target the particular case under observation.

A description of the specific reasons for the *false positive* result, and a discussion on possible solutions and research ideas, that will be studied in the next chapter, are provided in the following section.

4.4 Discussion

4.4.1 False positive results

An important IMS feature is that different applications may share some information to avoid duplicating functionalities. In this way, for example, the XDM server manages the address book of the user, and the Presence Server manages the user status and availability information. The PoC server, as well as other applications, makes use of both of these capabilities while managing its own sessions. Furthermore, the PoC server also provides features of the SIP notification framework for conference state [Roach 2002, Rosenberg 2006] in order to transmit information about the conference status to subscribed clients by means of `NOTIFY` messages. Due to this mix of features, most traces collected will contain packets for all different procedures performed by the server during the communication. For this reason, and since most of the communication in the IMS is done using SIP, the definition of invariants suitable for evaluation becomes more difficult.

Let us take as an example the case of invariant 4 previously defined. Considering the session sequence showed in the Figure 4.1, the invariant is correctly defined. Indeed, when finding an `OK` message, it indicates that a session initialization parameters established by an `INVITE` has been accepted. However the expected observation $OK(CSeq, From, To, CallID)$ is too general, and matches acceptance of other types of requests, such as `SUBSCRIBE`, `NOTIFY` or `BYE`, that are also found on the trace. This is shown in the following example.

Example 4.4.1. For invariant 4, on inspection of the trace shown in Figure 4.8, and starting from the end of the trace, upon observation of the last `OK` message, it will declare a match for the preamble of the invariant, and will start to look for an `INVITE` message, with $(CSeq_{(1)}, From_{(1)}, To_{(1)}, CallID_{(1)})$ as parameters. However, since the last `OK` message is a response to the `NOTIFY` request, as indicated by the parameters, then the evaluation of the invariant returns a FAIL result.

In case of controlled experiments, where the information in the traces is expected to be limited to a defined number of clients or sessions, manual analysis may separate

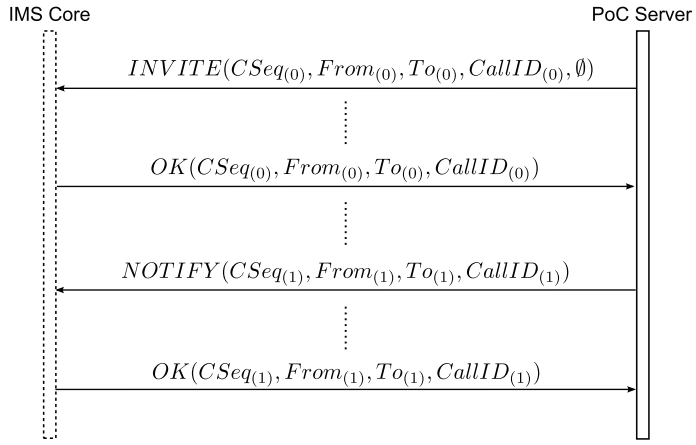


Figure 4.8: A trace including a `NOTIFY` – `OK` pair.

the expected results from the false positives. In the best case scenario, *false positive* results constitute noise in the results that should be avoided. However, in the worst case, such results may also be indicative that there are alternative behaviors not being tested. A discussion on alternative approaches and research ideas is provided in the following.

4.4.2 Motivating research ideas

In this particular case there is a simple solution, to further restrict the invariant, using the sub-values of the $CSeq$ header (as described for the invariant 6). The following formula

$$INVITE(<\text{`INVITE'}, Seq_{(0)}>, From_{(0)}, To_{(0)}, CallID_{(0)}, \emptyset)/?, *,$$
$$?/OK(<\text{`INVITE'}, Seq_{(0)}>, From_{(0)}, To_{(0)}, CallID_{(0)})$$

specifies that each time that an `OK` message is observed as a response to an `INVITE` message (defined by the `CSeq.method` field), then a corresponding `INVITE` must also appear in the trace. However, it is possible that due to the specificity of a given property, invalid behavior goes untested, leading to a *false negative*, or PASSED verdicts being provided for invalid behavior. One such case could occur, for instance, with invariant 6, where the evaluation of

$$?/INVITE(<\text{`INVITE'}, Seq_0>, User_0, \text{`ConferenceURI'}, CallID_0, \{User_1\}), *$$
$$OK(<\text{`INVITE'}, Seq_0>, User_0, \text{`ConferenceURI'}, CallID_0)/?, *,$$
$$?/ACK(<\text{`ACK'}, Seq_0>, User_0, \text{`ConferenceURI'}, CallID_0)$$

does not take into account that other SIP `2xx` responses (success responses including the `OK`, with code 200) also must be acknowledged. Even more, error responses to an INVITE, with codes 300 to 699, also require an `ACK` reply. However, testing such possibilities would require an invariant that can match alternative preamble sequences for a given output, which is not supported by the current invariant syntax.

Other point to be considered is that the current invariant syntax derived from the EFSM, $i_1/o_1, i_2/o_2, \ldots, i_n/o_n$, is not always suitable for testing some protocols, where the causal relations between inputs and outputs are not directly known, or where there may be multiple outputs for a single input (as with SIP). When capturing traces for system where multiple clients can interact, this becomes more critical, since the trace is simply a mix of multiple input and output packets, where a particular input i can only be related to an output o through the data parameters they contain.

In current invariant approaches [Ladani 2005], dealing with data requires extracting constraint information from the specification in order to evaluate constraints along control part. This approach limits somewhat the practicality of the invariant utilization, since a specification is required in order to evaluate data relations in the trace. It is interesting to explore the inverse approach then, invariants defined with control and data constraints, that can then be verified into the specification.

Finally, for IMS services, it might be interesting to test more complex requirements of the service, as described in the beginning of the chapter. For the PoC, for instance, when a client joins a conference, a notification must be sent to all clients subscribed for the '`conference`' event [Rosenberg 2006]. The following sequence of events should be matched to test such a feature

$$\begin{array}{ll} IF & subscribed(user_1, conference) \\ AND & joins(user_2, conference) \\ THEN & notify(user_1, user_2) \end{array}$$

where *subscribed* requires observing a `SUBSCRIBE` – `OK` pair, *joins* require observing `INVITE` – `OK` messages, and *notify*, requires observing a `NOTIFY` – `OK` pair.

All of these issues related to testing provide the motivation for the work described in the following.

4.5 Conclusion

In this chapter, we described our work for passive testing of IMS services, focusing particularly on a Push-to-talk Over Cellular (PoC) service implementation, as presented in [Lalanne 2009b, Lalanne 2009a]. This work was motivated by the fact that active testing techniques are not always possible to apply in testing of already deployed systems, particularly for proprietary implementations. In our work, an approach based on evaluation of invariants is presented, where six properties to test were chosen from the PoC specification document, in order to evaluate them into service traces.

Although the approach in general showed positive results, some *false positive* results were also found during the evaluation. Although these results are partly due to definition of properties considering only partial information from the specification documents, they can also be related to the limitations of the invariant approach for

evaluation of more complex traces, as those of IMS services, where multiple message exchanges can appear, and the only relation between inputs and outputs is through data contained in the messages.

Several motivating ideas, some of them related to the work in this thesis, are proposed as improvement to the invariant approach, dealing with definition of more general properties through the consideration of data relations, and the utilization of such an approach to test integration of features in IMS services. In the following chapter, several of these ideas will be developed, into what constitutes the main contribution of the present work.

Chapter 5

A Data-centric approach for Invariant Testing

Contents

5.1 Introduction

Passive testing is based on the observation of input and output events of an implementation under test in run-time. The term "passive" means that the tests do not disturb the natural run-time of a protocol as the implementation under test is not stimulated. The record of the event observation is called a *trace*. In order to check the conformance of the IUT, this trace will be compared to its expected behavior, defined either by a formal model (whenever available) or by one or more expected

functional properties (called invariants). The objective of passive testing is to provide a verdict about the conformance of an IUT, through the behavior observed in the trace.

Passive testing and runtime verification (or runtime monitoring) approaches are often conflated [Brzezinski 2009]. They both observe a run of the system (contained in a trace) and attempt to determine the satisfaction of a given correctness property [Leucker 2009]. However, while passive testing has the specific purpose to deliver a verdict with respect to the conformance of black-box implementations (IUT), verdicts in runtime verification are related to the satisfaction of any property that can be verified in a trace[1]. Generally, runtime verification approaches deal with the technical aspects of property evaluation and monitor generation, without necessarily attempting to provide a verdict about the system.

Both passive testing and runtime monitoring methodologies derive, respectively, from model-based testing and model checking techniques. Due to this, they are usually propositional in nature, that is, they are designed to focus primarily in the control part of the observation, and consider data only as an extension of the control part, usually by the addition of parameters. However, as previously described, such approach is not ideal when modern (e.g. application-layer) protocols are evaluated. It is the premise of the current work that a *message-based/data-centric* approach, i.e. a bottom-up definition of properties, starting from expected relations between message data fields to express properties of incremental complexity, provides a more effective solution for invariant-based testing of such protocols. This is inspired by the fact that events in a communication trace are usually collected in the form of *messages* or *packets*, i.e. collections of data, which are used for the extraction of both control and data parts.

In this chapter we present our approach for data-based invariant testing of protocol implementations. Invariants in our work are defined as a pair $I(\phi, \psi)$, where ϕ is a formula specifying a *test behavior*, i.e. a behavior that needs to be observed in the trace in order to provide a **pass** verdict. The formula ψ specifies a *conditional behavior*, i.e. the behavior that needs to be observed in the trace to provide a **fail** verdict when the evaluation of the test behavior does not produce conclusive information. The conditional behavior is required since lack of observation of the test behavior does not necessarily imply failure of the IUT to produce it. This issue is reflected in our semantics for satisfaction of formulas in the context of a trace, where evaluation of each formula can be determined within the truth values: '$\top$' (*true*), '$\bot$' (*false*), and '?' (*inconclusive*), respectively indicating that the formula is satisfied, not satisfied and that there is no sufficient information to determine satisfaction.

Several elements are presented as part of our approach: *i)* a formal definition of messages as collections of labelled data is provided, *ii)* a syntax/semantics for formulas is defined. The syntax provides a way to establish required behavior with respect

[1] The term *trace* in runtime verification is not restricted to observable events

to data relations between messages, instead of control parts. *iii)* An algorithm to determine satisfaction of a given formula in a trace is defined. *iv)* An algorithm to provide a verdict for an invariant, $I(\phi, \psi)$, with respect to the observations in the trace is also provided. Finally, some clarifying examples and experiments are also provided.

All of the concepts introduced here will be detailed in the following sections. Part of the work in this chapter was published [Lalanne 2011a] and a more complete version has been submitted to the *Information Processing Letters*[2] journal.

5.2 Motivation

Some of the following considerations regarding existing invariant-based techniques, a few of them introduced in previous chapters of this document, provide a motivation for the presented work.

Causality between inputs and outputs. Traces in the FSM and EFSM formalism, consist of a sequence of input/output pairs (i_j/o_j). However causal relations between inputs and outputs are not always evident in real traces. In many protocols, it is common to observe multiple outputs for a single input, or to expect several inputs before providing an output (as in the case of provisional responses in SIP).

When dealing with centralized protocols and services, this issue becomes even more critical, since observations in the trace may contain interactions between the IUT and multiple clients, making causal relations become even more difficult (or impossible) to determine based on control information alone.

Assumptions in invariant evaluation. For obligation invariants, the assumption or condition that the trace is "*long enough*" is made prior to evaluation. For instance, in the invariant $i/?, *, ?/\overline{o}$, it is required that the collection of the trace started from a point in the execution of the IUT before $i/?$ could be observed, otherwise it may not be possible to provide a verdict, since the trace may have been collected *too late*. Such assumption is not very strong in the case of a short invariant or for testing under controlled conditions, but it may not be valid otherwise.

For simple (or forward) invariants, as currently defined, such assumption is not required. For example, an invariant $i_1/?, *, i_2/o_2$ requires the observation of the input i_2 prior to the verification of o_2; if i_2 is not observed, then no verdict is given, therefore no further assumptions are necessary. However, as previously stated, causality between inputs and outputs may not always be established, therefore reducing the utility of the forward approach. If the input i_2 is allowed to be a wild-card character '?', as in $i_1/?, *, ?/o_2$, then the assumption that the trace is "*long enough*"

[2] http://www.journals.elsevier.com/information-processing-letters/

must also be made, otherwise the cases "the output o_2 was never produced", and "the output o_2 has not been observed yet", can not be distinguished.

The meaning of the trace being "*long enough*" is not necessarily clear in the general case. If the model for the IUT contains an initial state, then it suffices to identify the initial state in the trace to verify the assumption[3]. The same could be established, for the case of simple invariants, if the model for the IUT contains a final state. However, these requirements may not always hold, and the condition must be defined and verified case by case.

Evaluation based on data Assuming that causality of events (inputs/outputs) cannot be determined through control parts, then all causality must be determined through data. This provides the main motivation for our approach. Expressiveness of formulas is important, in order to avoid redundant evaluations, since the trace may be long. Succinctness of formulas is also desired, in order to make properties maintainable by the experts.

One further observation should be taken into account for the evaluation based on data. Let us suppose that the following forward invariant is being tested: $i(x)/?, *, ?/o(x)$. If the output $o(x+1)$ is captured in the trace, how can we distinguish the cases, "no output was returned" and "an output with the wrong data was returned"? Although both cases are indicative of a fault, the second case might be more indicative of the location of the fault in the implementation. Not much emphasis is given to this distinction in the present chapter, however is is important to consider when performing evaluation based on data.

Availability of the specification In general, invariant techniques require that the invariant is verified on the specification to determine its correctness. The specification is also required as part of the validation of the trace in [Bayse 2005], to determine the state of the IUT at the beginning of the trace. For constrained invariants [Ladani 2005], the specification is required to determine the constraints that will be evaluated in the trace along with the invariant.

Since a specification of the system is rarely available, particularly for large systems, requiring a specification for validation of the properties in the trace limits the practicality of the invariant approach. It is thus desirable to limit the utilization of the specification just for the verification of the properties, and use as much possible the information in the specification documents and from experts of the protocol for their definition.

[3] This is done in [Bayse 2005] through use of the specification, with UIO method or by a homing phase.

5.3 Preliminaries

5.3.1 Definitions

Invariants Let us detail on the definition of invariants from the work in [Bayse 2005] before introducing a generalization. Given a EFSM $M = (S, s_0, \mathcal{I}, \mathcal{O}, \vec{x}, T)$, a *forward invariant* is defined as $F(P, PI, TO)$, where

- P is a preamble sequence defined as $P ::= i/o, P \mid *, P \mid \varepsilon$, where $i \in \mathcal{I} \cup \{?\}$ is an input, $o \in \mathcal{O} \cup \{?\}$ and ε is the null sequence
- $PI \in \mathcal{I}$ is the preamble input.
- $TO \subseteq \mathcal{O}$ is the test output set.

The forward invariant is true if, whenever the sequence P is observed, then a pair PI/o (with $o \in TO$) is observed.

Similarly, a *backward invariant* is defined as $B(TS, TI, PO)$, where

- $TS ::= i/o, TS \mid *, TS \mid \varepsilon$ is the test sequence.
- $TI \in \mathcal{I} \cup \{?\}$ is the test input.
- $TO \subseteq \mathcal{O}$ is the test output.

The backward invariant is true if, whenever the I/O pair TI/o (with $o \in TO$) is observed on the trace, then it is preceded by the sequence TS.

Event-driven Extended Finite State Machine Given that causality between inputs and outputs cannot be assumed, then a model more suitable with the observations in a trace is the *Event-driven EFSM* (EEFSM)

Definition 5.3.1. An *Event-driven Extended Finite State Machine* (EEFSM) is a 6-tuple $(S, s_0, \Sigma, \vec{x}, \vec{y}, T)$, where

- S is a finite set of states with $s_0 \in S$ as the initial state
- Σ is a finite set of events, where for each $e(\vec{y}) \in \Sigma$, e is the event name and $\vec{y} = (y_1, \ldots, y_q)$ is a vector of event parameters
- $\vec{x} = (x_1, \ldots, x_p)$ is variable vector
- T is a finite set of transitions

Each transition $t \in T$ is a 7-tuple $(s, s', e(\vec{y}), P(\vec{x}, \vec{y}), A(\vec{x}, \vec{y}))$, where

- $s \in S$ is the beginning state of the transition
- $s' \in S$ is the ending state of the transition
- $e(\vec{y}) \in \Sigma$ is the triggering event
- $P(\vec{x}, \vec{y})$ is a predicate
- $A(\vec{x}, \vec{y})$ is an action

When the machine is in a current state $s \in S$ with internal variable values $\vec{x}_s$, upon reception of event e with parameters $\vec{y}_e$, it will follow the transition $(s, s', e(\vec{y}_e), P, A)$ if the predicate $P(\vec{x}_s, \vec{y}_e)$ holds. In such case the machine will update the internal variables by the action $\vec{x} = A(\vec{x}_s, \vec{y}_e)$ and finally leave the machine in state s'.

A sequence $e_1(\vec{y}_1), e_2(\vec{y}_2), \ldots, e_n(\vec{y}_n)$ is a **trace** for an EEFSM $M = (S, s_0, \Sigma, \vec{x}, \vec{y}, T)$ if there exist states $s, s_1, \ldots, s_{n-1}, s' \in S$, such that the following sequence of transitions is possible in M: $s \xrightarrow{e_1(\vec{y}_1);P_1(\vec{x}_0,\vec{y}_1);A_1(\vec{x}_0,\vec{y}_1)} s_1, \ldots, s_{n-1} \xrightarrow{e_n(\vec{y}_n);P_1(\vec{x}_{n-1},\vec{y}_n);A_n(\vec{x}_{n-1},\vec{y}_n)} s'$, where $\vec{x}_0$ is the internal variable state before the first transition and $P_1, \ldots, P_n$ and $A_1, \ldots, A_n$ are the respective predicates and actions for each transition.

Messages on real traces When observing a real traces, the collected information is in the form of *messages* or packets. A message is a collection of data as exchanged by the peers of the communication, with a format defined by the protocol specification. Both the control and data parts of the messages for testing in the specification are extracted from such messages. A formal definition of messages will be provided later in this chapter, however, for the rest of the work, the following proposition is used.

Proposition 5.3.1. Let the EEFSM $M = (S, s_0, \Sigma, \vec{x}, \vec{y}, T)$, be a model for an implementation of the protocol $\mathcal{P}$. Let $\mathcal{M}_\mathcal{P}$ be the set of all messages allowed by the protocol. It is assumed that a function $\gamma : \mathcal{M}_\mathcal{P} \to \Sigma$ exists, where for every message $m \in \mathcal{M}_\mathcal{P}$, $\gamma(m)$ maps a message to its control part. This function is called the *control function* of a protocol $\mathcal{P}$.

5.3.2 Preliminary Analysis

Using formulas to describe invariants For the previous definition of invariants, let us assume that for each combination of inputs and outputs (such as those defined by P, PI and TO), there exists an equivalent formula ϕ, in some logic, such that for a given combination T, the evaluation of its corresponding formula ϕ_T in a trace is '$\top$' (*true*), if and only if T is observed in the trace. Using this equivalence, then the following combinations are possible, taking into account wild-cards and null sequences.

- The observation an input followed by optional outputs, PI/TO, can be determined by the evaluation of the formula $\phi_{PI} \rightarrow \phi_{TO}$, with ϕ_{PI}, ϕ_{TO} the corresponding formulas for PI and TO

- The observation of $P, PI/TO$, an input/output sequence followed by an input and alternative outputs, can be determined by the evaluation of the formula $\phi_P \wedge \phi_{PI} \rightarrow \phi_{TO}$.

In backwards invariants the alternatives are

- The observation of TO, a set of outputs, can be determined by evaluation of the formula ϕ_{TO}

- The observation of TI/TO an input and a set of possible outputs, can be determined by $\phi_{TI} \wedge \phi_{TO}$

- The observation of TS, TO a set of possible outputs and a preceding input/output sequence is defined by: $\phi_{TO} \xrightarrow{before} \phi_{TS}$, where $\xrightarrow{before}$ is only used to denote that the formula ϕ_{TS} must be satisfied at a preceding point in the trace than ϕ_{TO}

- The observation $TS, TI/TO$, an input, a set of possible outputs and a preceding input/output sequence, is defined by: $\phi_{TI} \wedge \phi_{TO} \xrightarrow{before} \phi_{TS}$

If causality is removed, it does not make sense to differentiate between inputs and output sequences, then the available combinations are reduced to a preamble formula ϕ_P and a test formula ϕ_T. The range of formulas is then simplified to:

- ϕ_T, for evaluating a test formula on a trace

- $\phi_P \rightarrow \phi_T$ for forward evaluation

- $\phi_P \xrightarrow{before} \phi_T$ for backwards evaluation.

If the semantics of formulas includes the semantics of operators '$\rightarrow$' and '$\xrightarrow{before}$', i.e. forwards and backwards implication, then the behavior to be observed may be defined by the satisfaction of a general formula ϕ_T. Expressiveness, temporal precedence of evaluation and causality based on data will all depend on the syntax used to define ϕ_T. Formulas defined this way provide a generalization of the traditional forward/backwards invariants, allowing also to define more expressive properties depending on the chosen syntax.

Conditions on the trace As previously described, when causality is removed, the condition that the trace is "*long enough*" must be validated along with the invariant. If a specification is not available then three different strategies are possible:

- Assume that the trace is never long enough. In other words, if the test behavior defined by a formula ϕ_T is not observed, then nothing can be said about the validity of the trace and an **inconclusive** result must be returned. Although this alternative may not provide useful information in most cases, analysis of the verdicts might. For instance, if the number of **inconclusive** verdicts for a formula is significant, then this may be indication of a fault in the trace.
- Assume that the trace is long enough. If the test behavior (ϕ_T) is not observed, then it is an indication of a fault and a **fail** must always be returned. This might not be the correct strategy in every case, however it may be a safe assumption if enough events in the trace have been analyzed. It might however produce some *false positive* results in border cases.
- For forward and backwards invariants, a formula ϕ_C might be used as a condition to distinguish between **inconclusive** and **fail** verdicts. If while evaluating the test formula ϕ_T, the sequence defined by the formula ϕ_C is observed instead, then a **fail** verdict is returned, otherwise the result is inconclusive. Unfortunately, the formula ϕ_C is not guaranteed to exist, and since causality with the observations specified by ϕ_T is also a requirement, it may not always be simple to define.

Taking into account these strategies, an invariant is defined as follows.

Definition 5.3.2. Let ϕ and ψ be formulas (in some logic) identifying sequences of messages. An *invariant* is defined as the pair $I(\phi, \psi)$, where ϕ represents the expected or *test behavior* and ψ represents the *conditional behavior*. The evaluation of an invariant $I(\phi, \psi)$ must return the verdict **pass** if the behavior defined by ϕ is observed, **fail** if the behavior specified by ϕ is not observed and the behavior specified by ψ is observed, and **inconclusive** if none are observed.

As a convention, $I(\phi, \top)$ represents an invariant where the condition is *true*, i.e. a **fail** verdict must be returned if the sequence defined by ϕ is not observed. $I(\phi, \bot)$ represents an invariant where the condition is *false*, i.e. an **inconclusive** verdict must be returned if the sequence defined by ϕ is not observed.

Positive and negative properties Invariants, with the new definition introduced, are designed to validate *expected* behavior on a trace (through formulas), and failure to observe such behavior is used as indication of a fault. In this sense, the invariant represents a *positive property*, since it defines an expected behavior.

While the issue with testing on a finite trace is the difficulty to distinguish between lack of observation of a behavior and failure of the implementation to

produce it, determining satisfaction of a property does not share the same issues. An interesting alternative is then to test for *negative properties*, i.e. attempt to directly observe erroneous behavior. A *negative invariant* is defined in the following.

Definition 5.3.3. Let ϕ and ψ be formulas (in some logic) identifying sequences of messages. A *negative invariant* is defined as $I^-(\phi, \psi)$, where ϕ represents the negative *test* behavior and ψ is the *conditional* behavior. The evaluation of an invariant $I^-(\phi, \psi)$ must return the verdict **fail** if the behavior defined by ϕ is observed, **pass** if the behavior specified by ϕ is not observed and the behavior specified by ψ is observed, and **inconclusive** if none are observed. Analogously to positive invariants, $I^-(\phi, \top)$ returns a **pass** verdict if ϕ is **not** observed, and $I^-(\phi, \bot)$ returns an **inconclusive** verdict if ϕ is **not** observed.

5.4 Details of the proposed approach

As described in the preliminary analysis, testing of an invariant $I(\phi, \psi)$ in our approach requires establishing whether the sequences of messages defined by ϕ and ψ are present in the trace. This requirement closely relates to the objectives of runtime verification and concepts from some works in that area have been used for our approach [Halle 2008, Stolz 2008, Bauer 2007a]. Some more detailed information on these works and the relation to ours is provided in Section 5.7.

In what follows, we describe the different aspects related to the definition of the syntax, semantics and evaluation of properties for our approach. In our work, events on the trace are messages, therefore the syntax is designed to specify expected relations between messages and message data fields, as well as temporal precedence between messages. Control parts are not considered as part of the definition, however through the assumption introduced in Proposition 5.3.1, messages can be related to their control part. Many of the ideas presented here are have a basis on our earlier work for passive testing based with first-order logic [Lalanne 2011b].

5.4.1 Formal definition of protocol messages

In network protocols, communication peers decide the course of action on the basis of two things: locally stored state information (including internal data), and data contained in received messages from different peers. In passive testing, state information and internal data are unknown and only the message data is available to the tester. In the following, a definition of messages as data structures is provided.

A message in a communication protocol is, using the most general possible view, a collection of data fields belonging to multiple domains. For a given protocol, the *format* of the message, the way the data is arranged and grouped in order to be parsed, and the domains of the different data fields are defined by the requirements specification of the protocol.

Data fields in messages, are usually either *atomic*, i.e. the information they provide comes from using their value as a whole (e.g. *timestamp*, *packet number*, *name*, *port*), or *compound*, i.e. they are composed of multiple elements (e.g. an URI `sip:name@domain.org`). Due to this, we also divide the types of domains in *atomic* or *compound*.

In *atomic* data domains each element is an *atomic value*, in our work limited to numeric or string values[4]. In *compound* data domains each element (or *compound value*) is represented by a pair $(label, value)$, where *label* is used to indicate the functionality of the piece of data contained in *value*. An example is given in the following to clarify.

Example 5.4.1. The URI `sip:name@domain.org` can be represented by the compound value

$$\{(protocol, \text{`}\texttt{sip}\text{'}), (user, \text{`}\texttt{name}\text{'}), (domain, \text{`}\texttt{domain.org}\text{'})\}$$

where *protocol*, *user* and *domain* are labels that indicate the functionality of the atomic values '`sip`', '`name`' and '`domain.org`', respectively.

Formal definitions of compound value and compound domain are provided below.

Definition 5.4.1. Let $L = \{l_1, \ldots, l_k\}$ be a set of labels and $D_1, \ldots, D_k$ a sequence of data domains (not necessarily disjoint), with $k > 0$. A *compound value* of length k is a set $\{(l_i, v_i) \mid \forall i = 1 \ldots k \wedge v_i \in D_i \cup \{\varepsilon\}\}$.

A *compound domain* is then the set of all compound values with the same set of labels L and sequence of domains $D_1, \ldots, D_k$. Notice that for a given domain, all elements of the domain must have length k, however undefined elements are defined by using the null value represented by ε.

Definition 5.4.2. A *compound domain* is defined by the labels and domains of its elements. Given L a set of labels and $D_1, \ldots, D_k$, data domains (not necessarily disjoint), with $k > 0$. A compound domain $\mathcal{C}$ is represented by the $(k+1)$-tuple $(L, D_1, \ldots, D_k)$. Each element $v \in \mathcal{C}$ is a compound value $\{(l_i, v_i) \mid 1 \leq i \leq k\}$ with labels $l_i \in L$ and values $v_i \in D_i \cup \{\varepsilon\}$. Each D_i is called an *element domain*.

Definition 5.4.3. Let $\mathcal{C} = (L, D_1, \ldots, D_k)$ be a compound domain. Then a function $\delta_{\mathcal{C}} : \mathcal{C} \times L \longrightarrow \cup_{i=1}^{k} D_i \cup \{\varepsilon\}$ is defined, where given a compound value $v \in \mathcal{C}$ then, for each pair $(l_i, v_i) \in v$, $v_i = \delta_C(v, l_i)$.

It should be observed that element domains D_i are not required to be atomic domains, making it possible to define recursive structures by using compound domains within compound domains.

[4]Without loss of generality we restrict to numeric and string values, although the approach does not prevent treating other domains, for instance dates, as atomic.

Finally, given a network protocol $\mathcal{P}$, a compound domain $\mathcal{M}_{\mathcal{P}}$ is assumed to exist, where the set of labels and element domains derive from the message format defined in the protocol specification. A *message* of a protocol $\mathcal{P}$ is any element $m \in \mathcal{M}_{\mathcal{P}}$, that is, a message is any data value which is valid with respect to the protocol specification.

Example 5.4.2. A possible message for the SIP protocol, specified using the previous definition is

$$\begin{aligned} m = \quad & \{(method, \text{`INVITE'}), \\ & (status, \varepsilon), \\ & (from, \text{`alice@domain.org'}), \\ & (to, \text{`bob@domain.org'}), \\ & (cseq, \{(seq, 10), (method, \text{`INVITE'})\})\} \end{aligned}$$

representing an INVITE request from alice@domain.org to bob@domain.org. Notice that the value associated to the label *cseq* is also a compound value, $\{(seq, 10), (method, \text{`INVITE'})\}$.

In the example, given the message $m \in \mathcal{M}$, the domain of all SIP messages, it might be desirable to extract the value associated with the label *method* inside the value associated with *cseq*. This would require the function call $\delta_{cseq}(\delta_{\mathcal{M}}(m, cseq), method)$[5]. Accessing data inside messages is a basic requirement for the current approach. In order to simplify this type of nested calls, the function Δ is defined.

Definition 5.4.4. Given a compound value v, let $dom(v)$ denote the domain set of v. Let $\mathcal{L}$ be a sequence $(l_1, l_2, \ldots, l_n)$ of labels, $pop(\mathcal{L})$ a function that returns and removes the first element of the sequence and $len(\mathcal{L})$ a function that returns the length of the sequence[6]. The function Δ is then defined recursively as:

$$\Delta(v, \mathcal{L}) = \begin{cases} v & \text{if } len(\mathcal{L}) = 0 \\ \Delta(v', \mathcal{L}) & \text{if } v' = \delta_{dom(v)}(v, pop(\mathcal{L})) \text{ is a compound value} \\ \varepsilon & \text{otherwise} \end{cases}$$

Intuitively, the function Δ receives a compound value, and a sequence of labels and returns the value pointed by the labels, or ε (null) if the pointed value does not exist. In order to ease the reading of formulas in the rest of the paper, the notation $v.l_1.l_2.\ldots.l_n$ is used to represent the call $\Delta(v, (l_1, l_2, \ldots, l_n))$. For instance for the message defined in Example 5.4.2, the value accompanying the *method* label for *cseq* element in message m, would be represented by '*m.cseq.method*'. This is the notation that will be used through the rest of the paper.

[5] Since the value accompanying *cseq* in m is also a compound value of some domain $\mathcal{C}_{cseq}$, we use δ_{cseq} to indicate $\delta_{\mathcal{C}_{cseq}}$.

[6] The names of functions *pop* and *len* come from their analogues in the *stack* data structure.

5.4.2 Traces

A *trace* is a collection of messages of the same domain (i.e. using the same protocol) containing the observed interactions of an entity of a network (the point of observation) with one or more peers during an indeterminate period of time. In other words, the trace is the collection of all messages exchanged by the P.O within its life. Depending on the interpretation of life, such definition makes a trace potentially infinite. Testing of properties, however, can only occur in a finite segment of the trace.

Definition 5.4.5. Given the domain of messages $\mathcal{M}_\mathcal{P}$ for a protocol $\mathcal{P}$. A *trace* is a sequence $\Gamma = m_1, m_2, \ldots$ of potentially infinite length, where $m_i \in \mathcal{M}_\mathcal{P}$.

Definition 5.4.6. Given a trace $\Gamma = m_1, m_2, \ldots$, a *trace segment* is any finite subsequence of Γ, that is, any sequence of messages $\rho = m_i, m_{i+1}, \ldots, m_{j-1}, m_j$ $(j > i)$, where ρ is completely contained in Γ (same messages in the same order).

The order relations $\{<, >\}$ are defined in a trace, where for $m, m' \in \rho$, $m < m' \Leftrightarrow pos(m) < pos(m')$ and $m > m' \Leftrightarrow pos(m) > pos(m')$ and $pos(m) = i$, the position of m in ρ $(i \in \{1, \ldots, len(\rho)\})$.

In practical terms, the trace extraction process should assign each collected message to its position and time of observation, therefore the function *pos* will simply return that value. Since it is only possible to capture trace segments, in the rest of the document, *trace* will be used to refer to a trace segment, unless otherwise specified.

5.4.3 A Horn-based logic to express data-aware properties

5.4.3.1 Syntax

In order to describe properties, a syntax based on Horn clauses is used. The syntax is closely related to that of the query language Datalog, described in [Abiteboul 1995], for deductive databases. Formulas in this logic can be defined with the introduction of terms and atoms. A *term* is either a constant, a variable or a pointer to a sub-element of a variable, as obtained with the Δ function.

Definition 5.4.7. A *term* is either a constant, a variable or a *selector variable*. In Backus-Naur form (BNF):

$$t ::= c|x|x.l.l\ldots l$$

where c is a constant in some domain (e.g. a message in a trace), x is a variable, l represents a label, and $x.l.l\ldots l$ is called a *selector variable*, equivalent to evaluating $\Delta(x, < l, l, \ldots, l >)$ from Definition 5.4.4.

Definition 5.4.8. An *atom* is defined as

$$\begin{aligned} A ::= \quad & p(\overbrace{t, \ldots, t}^{k}) \\ | \quad & t = t \\ | \quad & t \neq t \\ | \quad & t < t \end{aligned}$$

where t represents a term, $p(t, \ldots, t)$ is a predicate of label p and arity k. The symbols $=$, $\neq$ and $<$ respectively represent the binary relations "equals to", "not equals to" and "lower than" (numeric comparison).

In this logic, relations between terms and atoms are stated by the definition of clauses. A *clause* is an expression of the form

$$A_0 \leftarrow A_1 \wedge \ldots \wedge A_n$$

where $A_0 = p(t_1^*, \ldots, t_k^*)$, called the *head* of the clause, defines a predicate of label p and arity k. Elements of the predicate (t_i^*) are called *head terms*, and restrict general *terms* for the head of the clause, where each head term can only be constant or a variable ($t^* ::= c|x$). The expression $A_1 \wedge \ldots \wedge A_n$, is called the body of the clause, where A_i are atoms. Notice that, in the clause definition, if $k = 0$ the predicate becomes a proposition (a predicate with no parameters). The body of the clause can also be empty and in that case, the clause represents a simple relation between the concrete terms in the head atom.

Example 5.4.3. The clause *sibling*('`alice`', '`bob`') specifies a relation only between "elements" *alice* and *bob*. The clause $sibling(x, y) \leftarrow parent(z, x) \wedge parent(z, y)$ establishes a generic sibling relation: "elements" sharing a parent.

Disjunction in this logic is provided by overloading predicate declarations, i.e. defining multiple clauses with the same head. The following set of declarations

$$\begin{gathered} A_0 \leftarrow A_{1,1} \wedge \ldots \wedge A_{1,n_1} \\ A_0 \leftarrow A_{2,1} \wedge \ldots \wedge A_{2,n_2} \\ \vdots \\ A_0 \leftarrow A_{p,1} \wedge \ldots \wedge A_{p,n_p} \end{gathered}$$

is equivalent to

$$A_0 \leftarrow (A_{1,1} \wedge \ldots \wedge A_{1,n_1}) \vee (A_{2,1} \wedge \ldots A_{2,n_2}) \vee \ldots \vee (A_{p,1} \wedge \ldots \wedge A_{p,n_p})$$

Example 5.4.4. The relation p, defined as follows

$$\begin{aligned} p(x) &\leftarrow x.method = \text{'}\texttt{REGISTER}\text{'} \\ p(x) &\leftarrow x.method = \text{'}\texttt{INVITE}\text{'} \end{aligned}$$

accepts all messages x with method `REGISTER` or `INVITE`.

Finally, a *formula* is defined as follows

- If $A_1, \ldots, A_n$ are atoms, with $n \geq 1$, then $(A_1 \wedge \ldots \wedge A_n)$ is a formula (an *atomic formula*).
- If ϕ and ψ are formulas, then so is $\phi \rightarrow \psi$.
- If x, y are variables and ϕ is a formula, then so are $(\forall_x \phi)$, $(\forall_{y>x} \phi)$, $(\forall_{y<x} \phi)$, $(\exists_x \phi)$, $(\exists_{y>x} \phi)$ and $(\exists_{y<x} \phi)$.

We can condense this information using the following EBNF:

$$\phi ::= A_1 \wedge \ldots \wedge A_n \mid \phi \rightarrow \phi \mid \forall_x \phi \mid \forall_{y>x} \phi \mid \forall_{y<x} \phi \mid \exists_x \phi \mid \exists_{y>x} \phi \mid \exists_{y<x} \phi$$

where $A_1, \ldots, A_n$ are atoms, $n \geq 1$ and x, y are variables. Some more details regarding the syntax are provided in the following.

- The $\rightarrow$ operator indicates causality in a formula, and should be read as "*if-then*" relation.
- The $\forall$ and $\exists$ quantifiers, are equivalent to its counterparts in predicate logic. However, and as it will be seen on the semantics of the logic, the quantifiers in this logic, only apply to the trace. Then, given a trace ρ, $\forall_x$ is equivalent to $\forall x \in \rho$ and $\forall_{y<x}$ is equivalent to $\forall y \in \rho; y < x$, with '$<$' indicating the order relation from Definition 5.4.6. These type of quantifiers are called *trace quantifiers* or *trace temporal quantifiers*.

5.4.3.2 Semantics

The semantics used on this work is related to the traditional Apt–Van Emdem–Kowalsky semantics for logic programs [Van Emden 1976], however some changes will be introduced to deal with messages and trace temporal quantifiers. We begin by introducing the concepts of substitutions (as described in [Nilsson 1990]) and of ground expressions.

Definition 5.4.9. A *substitution* is a finite set of bindings $\theta = \{x_1/t_1, \ldots, x_k/t_k\}$ where each t_i is a term and each x_i is a variable such that $x_i \neq t_i$ and $x_i \neq x_j$ if $i \neq j$.

The *application* $x\theta$ of a substitution θ to a variable x is defined as follows.

$$x\theta = \begin{cases} t & \text{if } x/t \in \theta \\ x & \text{otherwise} \end{cases}$$

The application of a substitution θ to a selector variable $x.l_1 \ldots l_k$ is defined as

$$x.l_1 \ldots l_k\theta = \begin{cases} t.l_1 \ldots l_k & \text{if } x/t \in \theta \text{ with } t \text{ a compound value} \\ x.l_1 \ldots l_k & \text{otherwise} \end{cases}$$

The application of a particular binding x/t to an expression E (atom, clause, formula) is the replacement of each occurrence of x by t in the expression. The application of a substitution θ on an expression E, denoted by $E\theta$ is the application of all bindings in θ to all terms appearing in E.

Example 5.4.5. The application of the substitution $\theta = \{x/1, y/2\}$ to the clause $C = p(x) \leftarrow q(x, y) \wedge r(y)$ is the clause $C\theta = p(1) \leftarrow q(1, 2) \wedge r(2)$.

Two substitutions can also be composed to form a new substitution. Let $\theta = \{x_1/t_1, \ldots, x_k/t_k\}$ and $\alpha = \{y_1/s_1, \ldots, y_l/s_l\}$ be substitutions. The composition of θ and α, denoted by $\theta\alpha$ is obtained from the set

$$\theta\alpha = \{x_1/(t_1\alpha), \ldots, x_k/(t_k\alpha), y_1/s_1, \ldots, y_l/s_l\}$$

by removing all $x_i/t_i\alpha$ where $x_i = t_i\alpha$ $(1 \leq i \leq k)$ and by removing those y_j/s_j for which $y_j \in \{x_1, \ldots, x_k\}$ $(1 \leq j \leq l)$. In other words, redundant bindings (x/x) and inconsistent bindings (there cannot be $x/1$ and $x/2$ simultaneously in the resulting substitution) are removed from the composed substitution.

Example 5.4.6. Given substitutions $\theta = \{x/y, z/1\}$ and $\alpha = \{y/2, z/3\}$, the composition of both substitutions is given by $\theta\alpha = \{x/2, y/2, z/1\}$.

Definition 5.4.10. A ground expression is any expression where only constant (ground) terms are present. A *ground instance* of an expression E is the expression $E\theta$, where θ is a substitution and every variable x_i in the expression E has a binding to a constant term t_i in θ.

Given $K = \{C_1, \ldots, C_p\}$ a set of clauses and $\rho = m_1, \ldots, m_n$ a trace. An *interpretation*[7] is any function I mapping an expression E that can be formed with elements (clauses, atoms, terms) of K and terms from ρ to one element of $\{\top, \bot\}$. It is said that E is true in I if $I(E) = \top$.

The **semantics of formulas** under a particular interpretation I, is given by the following rules.

- The expression $t_1 = t_2$ is true, iff t_1 equals t_2 (they are the same term).
- The expression $t_1 \neq t_2$ is true, iff t_1 is not equal to t_2 (they are not the same term).
- The expression $c_1 < c_2$ is true, iff c_1 and c_2 are numeric constants, and c_1 is lower than c_2
- A ground atom $A = p(c_1, \ldots, c_k)$ is true, iff $A \in I$.
- An atom A is true, iff every ground instance of A is true in I.

[7] Called a *Herbrand interpretation* in logic programming.

- The expression $A_1 \wedge \ldots \wedge A_n$, where A_i are atoms, is true, iff every A_i is true in I.
- A clause $C : A_0 \leftarrow B$ is true, iff every ground instance of C is true in I.
- A set of clauses $K = \{C_1, \ldots, C_p\}$ is true, iff every clause C_i is true in I.

An interpretation is called a *model* for a clause set $K = \{C_1, \ldots, C_p\}$ and a trace ρ if every $C_i \in K$ is true in I. A formula ϕ is true for a set K and a trace ρ (true in K, ρ, for short), if it is true in *every* model of K, ρ. It is a known result [Van Emden 1976] that if M is a *minimal* model[8] for K, ρ, then if $M(\phi) = \top$, then ϕ holds in K, ρ, denoted by $K, \rho \vDash \phi$.

The semantics of formulas is then defined as follows. Let K be a clause set, ρ a trace for a protocol and M a minimal model, the operator $\hat{M}$ defines the **semantics of formulas**.

$$\hat{M}(A_1 \wedge \ldots \wedge A_n) = \begin{cases} \top & \text{if } M(A_1 \wedge \ldots \wedge A_n) \\ \bot & \text{otherwise} \end{cases}$$

Then, for x and y variables in $\mathcal{M}_\mathcal{P}$ (the compound domain of a protocol $\mathcal{P}$), we define the **semantics of quantifiers** $\forall_x$ and $\exists_x$ for a potentially infinite trace $\Gamma = m_1, m_2, \ldots$ as

$$\hat{M}(\forall_x \phi) = \begin{cases} \top & \text{if } \hat{M}(\phi\theta) = \top, \forall\theta \text{ where } x/m \in \theta \text{ and } m \in \Gamma \\ \bot & \text{if } \exists\theta \text{ with } x/m \in \theta \text{ and } m \in \Gamma, \text{ where } \hat{M}(\phi\theta) = \bot \end{cases}$$

$$\hat{M}(\exists_x \phi) = \begin{cases} \top & \text{if } \exists\theta \text{ with } x/m \in \theta \text{ and } m \in \Gamma, \text{ where } \hat{M}(\phi\theta) = \top \\ \bot & \text{if } \hat{M}(\phi\theta) = \bot, \forall\theta \text{ where } x/m \in \theta \text{ and } m \in \Gamma \end{cases}$$

Since a finite trace ρ is a finite segment of an infinite execution, it is not possible to declare a '$\top$' result for $\forall_x \phi$, as in the infinite case, since we do not know if ϕ may become '$\bot$' after the end of ρ. Equivalently, for $\exists_x \phi$, it is unknown whether ϕ becomes true for future values of x. Similar issues have to be considered in passive testing (as described in the introduction to this chapter) as well as in runtime monitoring [Bauer 2006], for evaluations on finite traces. The semantics for trace quantifiers requires then the introduction of a new truth value '?' (inconclusive) to indicate that no definite response can be provided. The **semantics of quantifiers for finite traces** is defined as

$$\hat{M}(\forall_x \phi) = \begin{cases} \bot & \text{if } \exists\theta \text{ with } x/m \in \theta \text{ and } m \in \rho, \text{ where } \hat{M}(\phi\theta) = \bot \\ ? & \text{otherwise} \end{cases}$$

$$\hat{M}(\exists_x \phi) = \begin{cases} \top & \text{if } \exists\theta \text{ with } x/m \in \theta \text{ and } m \in \rho, \text{ where } \hat{M}(\phi\theta) = \top \\ ? & \text{otherwise} \end{cases}$$

[8] Obtained as $\cap M$, the intersection of all models for K, ρ

The rest of the quantifiers are detailed in the following, where x is assumed to be bound as a message previously obtained by $\forall_x$ or $\exists_x$

$$\hat{M}(\forall_{y>x}\phi) = \begin{cases} \bot & \text{if } \exists\theta \text{ with } y/m \in \theta, \text{ where } \hat{M}(\phi\theta) = \bot \text{ and } m > x \\ ? & \textit{otherwise} \end{cases}$$

$$\hat{M}(\exists_{y>x}\phi) = \begin{cases} \top & \text{if } \exists\theta \text{ with } y/m \in \theta, \text{ where } \hat{M}(\phi\theta) = \top \text{ and } m > x \\ ? & \textit{otherwise} \end{cases}$$

The semantics for $\forall_{y<x}$ and $\exists_{y<x}$ is equivalent to the last two formulas, exchanging $>$ by $<$. Finally, the truth value for $\hat{M}(\phi \rightarrow \psi) \equiv \hat{M}(\phi) \rightarrow \hat{M}(\psi)$, using the truth table shown in Table 5.1.

Table 5.1: 3-valued truth table for operator '$\rightarrow$'

ϕ	ψ	$\phi \rightarrow \psi$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\top$	?	?
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\top$
$\bot$	?	$\top$
?	$\top$	?
?	$\bot$	?
?	?	?

The semantics of formulas described in the current section is not meant to provide a method for procedural evaluation of formulas, since it would be quite inefficient to calculate every model of K and trace ρ in order to test a particular property. An algorithm for evaluation of formulas is provided in Section 5.4.5.

5.4.4 Example for the SIP protocol

Before providing the details of an evaluation algorithm, let us first clarify the concepts previously defined through an example for the SIP protocol (Chapter 3). The definitions provided here will also be useful later in this chapter.

For testing rules in the SIP protocol, a SIP message is defined with the fields described on Table 5.2. With such structure, the following clauses identify a message as a request or a response.

$$\begin{aligned} &\mathit{request}(x) \leftarrow x.\mathit{method} \neq \varepsilon \\ &\mathit{response}(x) \leftarrow x.\mathit{statusCode} \neq \varepsilon \end{aligned}$$

A message x is a response to another message y according to the following statement in RFC 3261 [Rosenberg 2002, section 8.2.6.2]

> The From field of the response MUST equal the From header field of the request. The Call-ID header field of the response MUST equal the Call-ID header field of the request. The CSeq header field of the response MUST equal the CSeq field of the request. The Via header field values in the response MUST equal the Via header field values in the request and MUST maintain the same ordering.
>
> If a request contained a To tag in the request, the To header field in the response MUST equal that of the request. However, if the To header field in the request did not contain a tag, the URI in the To header field in the response MUST equal the URI in the To header field.

which translates into the following predicate

$$\begin{aligned} responds(resp, req) \quad &\leftarrow \quad response(resp) \\ &\wedge \quad resp.from = req.from \\ &\wedge \quad comparable(req.to, resp.to) \\ &\wedge \quad resp.callId = req.callId \\ &\wedge \quad resp.cSeq.seq = req.cSeq.seq \\ &\wedge \quad resp.cSeq.method = req.cSeq.method \\ &\wedge \quad resp.via = req.via \end{aligned}$$

where *comparable* compares the `To` headers. If the `tag` parameter is defined, then it requires that both the `address` and `tag` fields are equivalent, otherwise it only verifies that the `address` part of the headers match. It is defined by the following disjunction

$$\begin{aligned} comparable(reqTo, respTo) \quad &\leftarrow \quad reqTo.tag = \varepsilon \\ &\wedge \quad reqTo.addr = respTo.addr \\ comparable(reqTo, respTo) \quad &\leftarrow \quad reqTo.tag = reqTo.tag \\ &\wedge \quad reqTo.addr = respTo.addr \end{aligned}$$

Using these definitions of clauses allows to express properties as the following:

- The property "*every message is either a request or a response*" can be tested defining the additional clauses

  $$sipMsg(x) \leftarrow request(x)$$
  $$sipMsg(x) \leftarrow response(x)$$

  then results for $\phi = \forall_x sipMsg(x)$ are answers to the property.

- The property "*every request must have a response after it*" is defined as

  $$\phi = \forall_x(request(x) \rightarrow \exists_{y>x} responds(y, x))$$

- The property "*every request except from `ACK` must have a response after it*" is defined as

  $$\phi = \forall_x(request(x) \wedge x.method \neq \text{`ACK'} \rightarrow \exists_{y>x} responds(y, x))$$

Table 5.2: Structure of a SIP message

Field	Label	Description
Method	method	Indicates the request type, i.e. `REGISTER`, `INVITE`, etc. If empty (ε) then message is a response.
Status-Code	statusCode	Numeric code for the status. If empty then the message is a request.
Request-URI	reqURI	Indicates the URI of the destination of the request (the invitee address or the location service address during registration).
From address	from.addr	URI indicating the sender of the message.
From tag	from.tag	Serves for dialog identification, it is defined by the UAC and it is always the same for all messages of a dialog.
To address	to.addr	URI indicating the recipient of the message.
To tag	to.tag	Indicates the remote dialog identifier, it is defined by the UAS upon reception of a dialog initiating request (e.g. `INVITE`). It can be null in the initiating request.
Call-ID	callId	Unique identifier to group series of messages.
CSeq	cSeq	Identifies the transaction the message belongs to with a sequence and a transaction originating method.
CSeq sequence	cSeq.seq	Sequence for the transaction.
CSeq method	cSeq.method	Method that originated the transaction.
Via	via	Indicates the transport and address where the responses in the transaction should be sent.

5.4.5 Evaluation of formulas

In Section 5.4.3.2, a two part semantics was defined for the logic: one for of formulas of type $A_1 \wedge \ldots \wedge A_n$ (atomic formulas), and a second one for formulas including trace quantifiers. In the current section we intend to provide an algorithm for the evaluation of formulas, which will also require a two part methodology: *1)* resolution of atomic formulas, where a variant of the classical SLD (Selective Linear Definite-clause) resolution algorithm [Apt 1982, Lloyd 1984] will be introduced. *2)* Evaluation of formulas (including quantifiers) and determination of satisfaction for a given trace. Both parts are described in the current section, although the most attention is given to the second part, given that the SLD algorithm is quite documented in the literature. We begin by introducing the concept of *unifiers* and *unification*.

Definition 5.4.11. Two terms t and s are *unifiable* if either t or s is a variable or they have the same value. A *unifier* of two expressions (Definition 5.4.10) E and F is a substitution θ such that $E\theta = F\theta$. If a unifier exists then the expressions are said to be *unifiable*. A substitution θ is said to be *more general* than a substitution σ if there exists a substitution α such that $\sigma = \theta\alpha$. A unifier θ is said to be the *most general unifier (mgu)* of two expressions E and F *iff* θ is more general than any other unifier. This is denoted as $\theta = mgu(E, F)$.

Example 5.4.7. Atoms $p(x, x)$ and $p(1, 2)$ are not unifiable, while the *mgu* of $p(x, y)$ and $p(1, 2)$ is the set $\{x/1, y/2\}$.

As a preliminary to the SLD-resolution algorithm we first define the function $unify$, that calculates the unification of two expressions, as follows

$$unify(A_1, A_2, \theta) := \begin{cases} true & \exists\alpha \textit{ substitution} \text{ such that } \alpha = mgu(A_1\theta, A_2\theta) \\ false & \text{otherwise} \end{cases}$$

where A_1 and A_2 are atoms, and θ is a substitution. If the atoms unify (result is $true$), the function $unify$ will also update the substitution given as argument with α (θ becomes $\theta\alpha$). As atoms are the basis to clauses, only unifications of atoms are considered for the resolution algorithm. The detailed procedure of unification is provided in the following.

The Procedure 5.1, iterates over all terms in the provided atoms and, for each pair of terms in the same position, it verifies that their bindings in θ and α unify. In the substitution α the algorithm keeps track of the result of unification of previous terms, and uses it to check subsequent iterations. For instance, in the unification of $p(x, x)$ and $p(1, 2)$ (using $\theta = \emptyset$), after the first iteration, α will contain the binding $x/1$, which will fail on second iteration when trying to unify it with the value 2.

Example 5.4.8. Calling $unify$ with arguments $A_1 = p(x, y)$, $A_2 = p(1, z)$ and $\theta = \{z/2\}$ will return $true$ and θ will become $\{z/2, x/1, y/2\}$. Using arguments $A_1 = p(x, y)$, $A_2 = p(1, z)$ and $\theta = \emptyset$, returns $true$ and $\theta = \{x/1, y/z\}$.

Procedure 5.1 $unify(A_1, A_2, \theta)$

Input: Atoms A_1, A_2. Substitution θ with current bindings.
Output: **true** if atoms unify and θ updated with new bindings.

```
1: if A1 and A2 have different predicate label or arity then
2:    return false
3: α ← ∅
4: for t ← A1[i] term i in A1 do
5:    s ← A2[i]
6:    if (tθ)α unifies with (sθ)α then
7:       if (tθ)α is a variable then
8:          α ← α ∪ {(tθ)α/(sθ)α}
9:       else                                  /* (sθ)α can only be a variable */
10:         α ← α ∪ {(sθ)α/(tθ)α}
11:   else
12:      return false
13: θ ← θα                          /* Compose θ with the result of the unification */
14: return true
```

5.4.5.1 SLD-resolution

In SLD-resolution, given a set of clauses $K = \{C_1, \ldots, C_n\}$, there exists a solution for a formula (also called query) $A_1 \wedge \ldots \wedge A_p$ if, for every atom A_i, there exists a clause $B_0 \leftarrow B_1 \wedge \ldots B_q$ in K where its head B_0 unifies with A_i (with a unifier θ), and that the resulting formula $(A_1 \wedge \ldots \wedge B_1 \wedge \ldots B_q \wedge \ldots \wedge A_p)\theta$, obtained after replacing A_i by $B_1 \wedge \ldots B_q$, also has a solution. If in the clause, $q = 0$ and the unification succeeds, then $A_i\theta$ is considered to be true and can be replaced by the symbol '$\top$' in the resulting formula. This can be summarized by the following inference rule (borrowed from [Nilsson 1990])

$$\frac{A_1 \wedge \ldots \wedge A_{i-1} \wedge A_i \wedge A_{i+1} \wedge \ldots \wedge A_p \quad B_0 \leftarrow B_1 \wedge \ldots B_q}{(A_1 \wedge \ldots \wedge A_{i-1} \wedge B_1 \wedge \ldots B_q \wedge A_{i+1} \wedge \ldots \wedge A_p)\theta}$$

where

1. $A_1, \ldots, A_p$ are atoms;
2. $B_0 \leftarrow B_1 \wedge \ldots B_q$ is a clause in K, with renamed variables, so no conflicts can occur if the formula and the clause coincide on the variable names;
3. $mgu(A_i, B_0) = \theta$.

In the present work, A_i can also be of type $t_1 = t_2$, $t_1 \neq t_2$ or $t_1 < t_2$. In this case the algorithm must evaluate the operation $t_1\theta = t_2\theta$, $t_1\theta \neq t_2\theta$[9] or $t_1\theta < t_2\theta$,

[9] If either $t_1\theta$ or $t_2\theta$ are variables, then $=$ is treated as unification ($x = a$ assigns a to x if x is not bound), however the evaluation of $\neq$ should return an error unless $t_1\theta$ and $t_2\theta$ are the same variable.

then replacing A_i in the formula by '$\top$' if the evaluation succeeds. If there exist more than one clause in K that can be unified with A_i, the algorithm must test all alternatives until finding one that replaces every atom in the resulting formula by '$\top$' (we cannot declare false until all alternatives are tested), this is also illustrated by the Figure 5.1(b) for Example 5.4.9.

As it can be observed in previous examples, the syntax of the logic allows for reuse of variable symbols, for instance, both a clause $p(x) \leftarrow B$ and a query $p(x)$ can be defined, although both x may not necessarily represent the same value. In order to perform unification, they must be treated as different variables. For this, a variable renaming step must also be performed. To evaluate the formula $p(x)$ we rename x to x_0, and when selecting a clause for unification, all x in the clause are replaced by x_1, this way, if the unification succeeds, the binding x_0/x_1 will be added to θ and different symbols are clearly distinguished.

A short example is provided below to illustrate the resolution mechanism.

Example 5.4.9. Let the following be a clause set for the SIP protocol, as defined in Section 5.4.4, simplified for the purposes of this example

$$sipMsg(x) \leftarrow request(x) \tag{5.1}$$
$$sipMsg(x) \leftarrow response(x) \tag{5.2}$$
$$request(x) \leftarrow x.method = \text{`}\texttt{INVITE}\text{'} \tag{5.3}$$
$$response(x) \leftarrow x.statusCode = 200 \tag{5.4}$$

The clause set indicates that a valid SIP message is either a request or a response. A valid request is one with method `INVITE` and a valid response is one with status code 200. Again, this is an oversimplification made for the purposes of the example. Let us test a property stating that only valid messages should be found in the trace, i.e. $\phi = \forall_x sipMsg(x)$. However, since the interest is to demonstrate SLD-resolution, for now we will ignore the quantifiers and assume that the value of x is already determined to be $x = m_i$, an arbitrary message in the trace. The Figure 5.1 shows the evaluation steps when the query is evaluated for messages m_i, a request and m_j, a response. Assuming that clauses are evaluated in the order they are defined, the resolution for m_i, follows the following steps

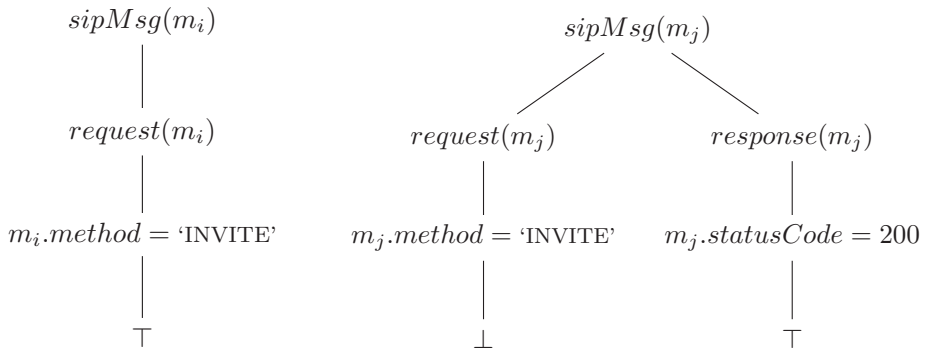
1. Look for a clause matching $sipMsg(x_0)$, i.e. with same predicate label and arity. This returns clauses 5.1 and 5.2;

2. Unify $sipMsg(x_0)$ with the head of clause 5.1, $sipMsg(x_1)$ using substitution $\theta = \{x_0/m_i\}$. This results in $\theta = \{x_0/m_i, x_1/x_0\}$;

3. Since unification succeeds, evaluate the body of the clause as a new query. Look for a clause matching $request(x_1)$, i.e. clause 5.3;

4. Unify $request(x_1)$ with the head of clause 5.3, $request(x_2)$ and substitution θ. This results in $\theta = \{x_0/m_i, x_1/x_0, x_2/x_1\}$;
5. Since unification succeeds, evaluate the body of the clause as a new query;
6. The only atom in the body is of type $t_1 = t_2$, then evaluate $x_2.method =$ '`INVITE`' using $\theta = \{x_0/m_i, x_1/x_0, x_2/x_1\}$;
7. Assuming that m_i is a request: evaluation is true, then $\top$ is returned.

Evaluation for m_j follows the same procedure, except that it fails on step 7 (since it is a response). As there is still one alternative to analyze from step 1, the resolution continues until the evaluation of $m_j.status = 200$ and $\top$ is returned. If all branches of the tree end with a '$\bot$', then $\bot$ is returned for the property.



(a) Assumming that $x = m_i$ is a request

(b) Assumming that $x = m_j$ is a response

Figure 5.1: Alternative resolution trees for query $sipMsg(x)$

The detailed SLD-resolution algorithm is provided in Procedure 5.2. The resolution starts with a formula $A_1 \wedge \ldots \wedge A_p$ in the form of a stack (A_1 at the top of the stack). For each atom on the stack it looks for a matching clause (a clause with the same predicate label and arity) and adds the body of the clause to the stack to recursively call solve. When the stack is empty, a solution has been found and it notifies it using the procedure $useSolution()$. Although it is not on the algorithm, an alternative to line 4 should also check whether A matches '$=$', '$\neq$' or '$<$', and respectively evaluate the equality, inequality or comparison.

5.4.5.2 Evaluating formulas in a trace

Given a formula ϕ, defined using a set of clauses K, it is not necessarily interesting to attempt to produce a single (general) satisfaction result ('$\top$', '$\bot$' or '?') for a particular trace ρ. Let us take, as example, a property $\forall_x(\exists_y p(x, y))$. Due to the

Procedure 5.2 $sldSolve(K, S, \theta)$

Input: Set of clauses K. Stack S containing the atoms remaining for evaluation. Substitution θ with the initial bindings.

Output: $\top$ if the formula has a solution.

```
1:  if S is not empty then
2:    A ← pop(S)                                  /* Remove first atom of the stack */
3:    solved ← ⊥
4:    for (B_0 ← B_1 ∧ ... ∧ B_q) ∈ K where B_0 matches with A do
5:      renameVars(B_0, B_1, ..., B_q)            /* Rename variables in the clause */
6:      α ← θ
7:      if unify(A_0, B_0, α) then
8:        if q > 0 then
9:          push({B_1, ..., B_q}, S)              /* B_1 is now at the top of S */
10:         solved ← sldSolve(S, α)               /* Solve the rest of the stack */
11:         pop({B_1, ..., B_q}, S)
12:       else                                    /* The clause does not have a body */
13:         solved ← sldSolve(S, α)
14:   push(A, S)                                  /* Put A back to the top of S */
15:   return solved
16: useSolution(θ)                                /* If S is empty, a solution has been found */
17: return ⊤
```

semantics of the logic (Section 5.4.3.2), the truth value of $\forall_x \phi$ can only be '$\bot$' if the value of ϕ is also '$\bot$', otherwise the value defaults to '?'. Since $\exists_y p(x_0, y)$ can never be '$\bot$' (for any x_0), then the evaluation of $\forall_x(\exists_y p(x, y))$, can never yield a result other than *inconclusive*. Even though results obtained this way do not provide useful information, particular values of (x, y) that make $p(x, y)$ true or false, do. Multiple results will be expected for the evaluation of a formula in a trace and two rules will be used for reporting a particular result.

1. Given a formula $\forall_x \phi$, an independent result should be declared for every value of x. For instance, given $\forall_x sipMsg(x)$, the evaluation $sipMsg(x_0)$ for every x_0 in the trace ρ should be provided as a result.

2. Given a formula $\exists_x \phi$, a result should be given only if it exists some value of x in the trace that makes the property true. Using an analogous example, for the property $\exists_x sipMsg(x)$, an algorithm should only report '$\top$' for the $x_0 \in \rho$ that makes $sipMsg(x_0)$ true. Any other values for x are irrelevant for the resolution.

A recursive algorithm is defined for evaluation, and a detailed description of the different cases in the evaluation is provided as follows, where $eval(\phi, \theta, \rho)$ returns the evaluation of the formula ϕ using substitution θ into trace ρ. The value of θ at the beginning of the evaluation is $\theta = \emptyset$.

- The evaluation of a formula $\forall_x \phi$ branches the evaluation of ϕ for each possible value of x in the trace. The result is provided by

$$eval(\forall_x \phi, \theta, \rho) = \begin{cases} eval(\phi, \alpha, \rho) & \forall m \in \rho \text{ where } \alpha = \theta \cup \{x/m\} \\ ? & \text{if no } \bot \text{ results were found} \end{cases}$$

- Evaluation of a formula $\exists_x$ looks for a '$\top$' result to the evaluation of ϕ.

$$eval(\exists_x \phi, \theta, \rho) = \begin{cases} \top & \text{if } \exists m \in \rho \text{ where } eval(\phi, \alpha, \rho) = \top \text{ with} \\ & \alpha = \theta \cup \{x/m\} \\ ? & \text{otherwise} \end{cases}$$

- Evaluation of a formula $\forall_{y>x} \phi$ assumes that a binding x/m_0 with $m_0 \in \rho$ already exists in the substitution θ and branches the evaluation of ϕ for every message after the position of m_0.

$$eval(\forall_{y>x} \phi, \theta, \rho) = \begin{cases} eval(\phi, \alpha, \rho) & \forall m \in \rho \text{ where } m > x\theta \text{ and } \alpha = \theta \cup \{y/m\} \\ ? & \text{if no } \bot \text{ results were found} \end{cases}$$

- Evaluation of a formula $\exists_{y>x}$ looks for a '$\top$' result to the evaluation of ϕ after the position of x in the substitution θ

$$eval(\exists_{y>x} \phi, \theta, \rho) = \begin{cases} \top & \text{if } \exists m \in \rho \text{ with } m > x\theta \text{ where } eval(\phi, \alpha, \rho) = \top \\ & \text{and } \alpha = \theta \cup \{y/m\} \\ ? & \text{otherwise} \end{cases}$$

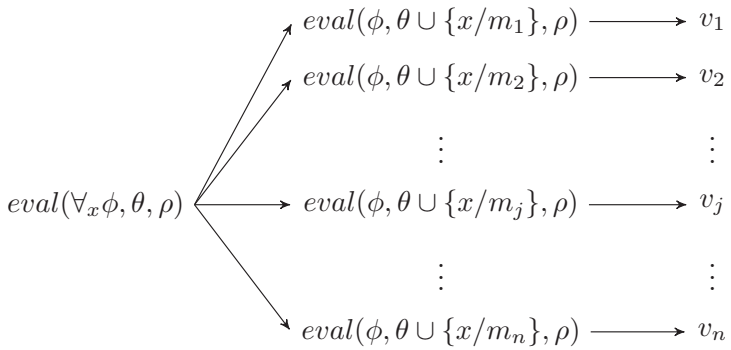
- Evaluation of $\forall_{y<x} \phi$ and $\exists_{y<x} \phi$ are analogous to their equivalents with '$>$' just replacing every occurrence of '$>$' by '$<$'.

- Evaluation of a formula $\phi \rightarrow \psi$ first will evaluate ϕ and if the result is '$\top$', then evaluates ψ. If the latter also has the value '$\top$', then the result of evaluation is '$\top$'. Any new bindings defined from the evaluation of ϕ must be used in the evaluation of ψ. If the evaluation of ϕ is '$\bot$' or '?', then the result is considered *vacuous* (uninteresting) [Fraser 2009, section 5.1], since $eval(\bot \rightarrow \psi, \theta, \rho) = \top$ and $eval(? \rightarrow \psi, \theta, \rho) = ?$, independently of the value of ψ. Both those cases are ignored during the evaluation.

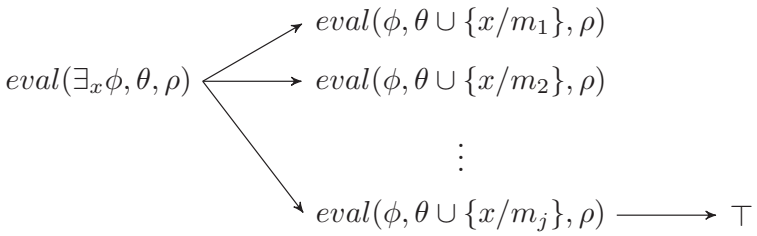
$$eval(\phi \rightarrow \psi, \theta, \rho) = \begin{cases} \top & \text{if } eval(\phi, \theta, \rho) = \top \text{ and } eval(\psi, \theta, \rho) = \top \\ \bot & \text{if } eval(\phi, \theta, \rho) = \top \text{ and } eval(\psi, \theta, \rho) = \bot \\ ? & \text{if } eval(\phi, \theta, \rho) = \top \text{ and } eval(\psi, \theta, \rho) = ? \end{cases}$$

- Evaluation of a formula $A_1 \wedge \ldots \wedge A_k$, where A_i are atoms, returns the value obtained using SLD-resolution, by using Procedure 5.2

$$eval(A_1 \wedge \ldots \wedge A_k, \theta, \rho) = \begin{cases} \top & \text{if } (A_1 \wedge \ldots \wedge A_k)\theta \text{ has a solution} \\ \bot & \text{otherwise} \end{cases}$$



(a) Evaluation of $\forall_x\phi$ returns '?' unless it exists $v_j = \bot$



(b) Evaluation of $\exists_x\phi$ returns '?' if no evaluation to '$\top$' of ϕ is found

Figure 5.2: Evaluation of formulas with quantifiers on a trace.

The previous rules define every possible case for the evaluation algorithm. Figure 5.2(a) shows the branching of the evaluation performed for a $\forall_x\phi$ formula. The evaluation of the $\forall$ quantifier splits the evaluation and one result is returned for each independent evaluation. A different case occurs with $\exists$, shown in 5.2(b), evaluation of the quantifier evaluates the sub-formula for each message in the trace until a '$\top$' result is found, which prompts the algorithm to return.

5.4.5.3 Complexity of the algorithm

As seen in the previous section, the evaluation of formulas can be represented by a tree. Intuitively, the time complexity of evaluation will depend on the number of nodes on the tree. The memory complexity, on the other hand, will only depend on the height of the tree, given that only the part of the tree being evaluated needs to be kept on memory (top-down resolution). Time complexity is then the most critical issue, therefore we will focus on that in the current section.

Since each evaluation will eventually arrive to one or more formulas of type $A_1 \wedge \ldots \wedge A_p$, two times can be recognized for each formula: $T_{eval}(\phi)$ represents the worst-case time of evaluation of a formula ϕ, and $T_{sld}(\psi)$ represents the time

required for the SLD evaluation of a formula $\psi = A_1 \wedge \ldots \wedge A_p$. Given a formula with k quantifiers $Q^1_{x_1} \cdots Q^k_{x_k}(A_1 \wedge \ldots \wedge A_p)$, where each $Q^j \in \{\forall, \exists\}$ and a trace $\rho = m_1, \ldots, m_n$, then the relation between T_{eval} and T_{sld} is described by:

$$\begin{aligned} &T_{eval}(Q^1_{x_1} \cdots Q^k_{x_k}(A_1 \wedge \ldots \wedge A_p)) \\ &\quad = \sum_{i_1=1}^{n} \cdots \sum_{i_k=1}^{n} T_{sld}((A_1 \wedge \ldots \wedge A_p)\theta_1 \cdots \theta_k) \end{aligned}$$

where $\theta_j = \{x_j/m_{i_j}\}$ is the substitution obtained by the evaluation of the quantifier $Q^j_{x_j}$. For a simple formula the height of the resolution tree is small compared with the length of the trace, therefore an upper bound for the SLD resolution time can be used inside the summation.

$$T_{sld}((A_1 \wedge \ldots \wedge A_p)\theta_1 \cdots \theta_k) \leq T, \theta_j = \{x_j/m_{i_j}\}, \forall i_1, \ldots, i_k$$

then, applying this inequality

$$\begin{aligned} &\sum_{i_1=1}^{n} \cdots \sum_{i_k=1}^{n} T_{sld}((A_1 \wedge \cdots \wedge A_p)\theta_1 \cdots \theta_k) \\ &\quad \leq \sum_{i_1=1}^{n} \cdots \sum_{i_k=1}^{n} T = n^k T \end{aligned}$$

which shows that the worst case complexity for this type of formula is $\mathcal{O}(n^k)$, where n is the length of the trace and k is the number of quantifiers in the formula. Although it seems large, it should be emphasized that it is the worst case complexity, i.e. the complexity for a trace where the evaluation of every quantifier returns '?'. It should also be clarified that this corresponds to the complexity of analyzing the whole trace, and not for obtaining individual solutions, which depends on the type of quantifiers used. For instance for a property $\forall_x p(x)$, individual results are obtained in $\mathcal{O}(1)$, and for a property $\forall_x \exists_y q(x, y)$, results are obtained in the worst case in $\mathcal{O}(n)$.

For a formula such as $\forall_x \exists_{y>x} p(x, y)$, the analysis of the complexity is similar, with the exception that the indexes on the summation will no longer be independent, using the same methodology, the time complexity is found to be

$$T_{eval}(\forall_x \exists_{y>x} p(x, y)) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} T_{sld}(p(x, y)\theta_i\theta_j) \leq \frac{1}{2}n^2 T - \frac{1}{2}nT$$

Similarly, for a formula with 3 quantifiers, the time complexity is given by $\frac{1}{6}n^3T - \frac{1}{2}n^2T + \frac{1}{3}nT$. Following an inductive methodology, it is easy to see that, for a formula $Q_{x_1} Q_{x_2>x_1} \cdots Q_{x_k>x_{k-1}}(A_1 \wedge \ldots \wedge A_p)$, the complexity is given by a polynomial of order k.

For a formula with a '$\rightarrow$' operator

$$\underbrace{Q \ldots Q}_{k}(\underbrace{Q \ldots Q}_{l}(A_1 \wedge \ldots \wedge A_p) \rightarrow \underbrace{Q \ldots Q}_{m}(A'_1 \wedge \ldots \wedge A'_q))$$

where Q represent different quantifiers, it is also simple to show that the time complexity of the evaluation is $\mathcal{O}(n^{k+max(l,m)})$ in the worst case.

The last result shows that defining formulas with a '$\rightarrow$' operator has advantages in terms of complexity. For instance, evaluation of the formula $\forall_x(\exists_y p(x,y) \rightarrow \exists_z q(z)$ has complexity $\mathcal{O}(n^2)$, while the formula $\forall_x \exists_y \exists_z (p(x,y) \wedge q(z))$ has complexity of $\mathcal{O}(n^3)$ in the worst case.

5.5 Evaluating invariants

Having defined a syntax and semantics for formulas, as well as an algorithm to evaluate them on the trace, the next step is to evaluate them as invariants. Let us consider the simplest cases first. Given a trace $\rho = m_1, m_2, \ldots, m_n$ and property ϕ.

- For a positive invariant with a true condition, $I(\phi, \top)$, the following steps are followed: *1)* call the procedure $eval(\phi, \theta = \emptyset, \rho)$ (Section 5.4.5.2), *2)* for every '?' or '$\bot$' result returned, emit a **fail** verdict. For every '$\top$' result, emit a **pass** verdict. *3)* Use the bindings in the resulting substitution θ to identify the messages involved in the failure.

- For a positive invariant with a false condition $I(\phi, \bot)$: *1)* call $eval(\phi, \theta = \emptyset, \rho)$ *2)* for every '$\bot$' result, return **fail** verdict, for every '$\top$' emit a **pass** verdict. for every '?' emit an **inconclusive** verdict. *3)* The bindings in the resulting substitution θ serve to identify the messages involved in the failure.

- For a negative invariant with a true condition $I^{-}(\phi, \top)$: *1)* call the procedure $eval(\phi, \theta = \emptyset, \rho)$, *2)* for every '$\top$' result returned, emit a **fail** verdict. For every '$\bot$' or '?' result, emit a **pass** verdict, *3)* The bindings in θ serve to identify the messages involved in the failure.

- For a negative invariant with a false condition $I^{-}(\phi, \bot)$: *1)* call the procedure $eval(\phi, \theta = \emptyset, \rho)$, *2)* for every '$\top$' result returned, emit a **fail** verdict. For every '$\bot$' result emit a **pass** verdict, for every '?' result, emit an **inconclusive** verdict, *3)* The bindings in θ serve to identify the messages involved in the failure.

For an invariant $I(\phi, \psi)$ (or its negative version), the conditional behavior defined by ψ must also be evaluated if the evaluation of ϕ returns '?'. It is not necessary though, to evaluate ψ in the whole trace, but only on the last segment checked during the evaluation of ϕ. The evaluation of the conditional part of an invariant is described in the following.

Given an invariant $I(\phi, \psi)$ and a trace ρ of length n, the following results are expected from the application of $eval()$ on ϕ

- A result '$\top$', '$\bot$' or '?' of the evaluation.

- A substitution θ with the bindings at the end of the evaluation.
- An interval $[b, e]$, with $b \geq 1$ and $e \leq n$ indicating the last range of messages reviewed during the evaluation of ϕ.

Let $\rho_{[b,e]}$ denote the segment of the trace ρ starting from the message in the position b to the message in the position e. The final step in the evaluation of an invariant $I(\phi, \psi)$ or $I^{-}(\phi, \psi)$, upon obtaining an '?' result for the evaluation of ϕ is defined by:

1. Call the procedure $eval(\psi, \theta, \rho_{[b,e]})$, where θ is the substitution resulting from evaluating ϕ.
2. In the positive version of the invariant, if the result of the evaluation of the condition ψ is '$\top$', then return a **fail** verdict, otherwise return an **inconclusive** verdict.
3. In the negative case, if the result of the evaluation of ψ is true, then return a **pass** verdict, otherwise return an **inconclusive** verdict.

5.6 Experiments

The syntax and evaluation algorithms described previously have been implemented into a framework, briefly described in Appendix A. In this section we provide some experiments, in order to show real examples of invariant definitions and the capabilities of the algorithm to provide conformance verdicts. The configuration of the framework for the experiments is provided in Appendix B and the implementation is available at the address `http://www-public.int-evry.fr/~lalanne/thesis.html`.

For the experiments, the traces used in Chapter 4 were used. These traces contain communication between the client and the PoC server, including registrations, PoC exchanges and subscriptions to presence information. Also, many packets for protocols different than SIP (TCP, RTCP, TalkBurst) appear as well. Although the tool provides a means for filtering such type of messages, due to the massive amount of extra information (in one case, from 137530 messages, only 299 were SIP messages), filtering was done prior to the tests. In the following subsections, the invariants used for evaluation, the syntax and the obtained results for each one are described.

For every request there must be a response

This property can be used for a monitoring purpose, in order to draw further conclusions from the results. Due to the nature of the property, *false* results can never

be provided for the evaluation of the test part of the invariant. Furthermore, given the generality of the test part of the invariant, a condition cannot be defined, since the condition depends on the type of request and response. Finally, due to the fact that the provided traces are not very long, a '$\perp$' condition is used to avoid *false positive* verdicts (cf. Section 5.3.2).

Nevertheless, as it will be shown, *inconclusive* results can also provide interesting information about the peers of the communication. The invariant $I(\phi_1, \psi_1)$ is defined with the following ϕ_1, ψ_1:

$$\begin{aligned} \phi_1 = \quad & \forall_x(request(x) \wedge x.method \neq \text{`ACK'} \\ & \quad \rightarrow \exists_{y>x}(nonProvisional(y) \wedge responds(y, x))) \\ \psi_1 = \quad & \perp \end{aligned}$$

where $nonProvisional(x)$ accepts all non provisional responses (responses with status ≥ 200) to requests with method different than `ACK`, which does not require a response. The results from the evaluation on the traces is shown on Table 5.3. As expected, most traces show only true results for the property evaluation, however traces 4 and 10 show an unusual number of inconclusive results. Taking a closer look at both traces, all of the errors correspond to `NOTIFY` messages from the PoC server to the client for the *conference* event (RFC 4575). Many of the requests are retransmissions and all of the events occur at the end of trace, which is an indication that the client closed the connection before receiving the `NOTIFY` message.

Table 5.3: Results of testing the property "*for every request there must be a response*" on the set of traces.

Trace	No. of messages	**pass**	**fail**	**inconclusive**	Time (s)
1	31	6	0	0	0.556
2	62	24	0	0	0.552
3	126	48	0	0	0.423
4	141	55	0	9	0.48
5	189	99	0	0	0.809
6	190	78	0	0	0.504
7	214	93	0	0	0.352
8	331	151	0	0	0.699
9	409	206	0	0	0.985
10	625	281	0	14	1.457

Every session initialization must be acknowledged

As described in Chapter 3, the session initialization procedure is a three-way handshake, composed by the messages `INVITE` – `OK` – `ACK`. The construction of the `ACK` request is detailed in [Rosenberg 2002, section 13.2.2.4]

> The UAC core MUST generate an ACK request for each 2xx received from the transaction layer. The header fields of the ACK are constructed in the same way as for any request sent within a dialog (see Section 12) with the exception of the CSeq and the header fields related to authentication. The sequence number of the CSeq header field MUST be the same as the INVITE being acknowledged, but the CSeq method MUST be ACK. The ACK MUST contain the same credentials as the INVITE.

The complete `ACK` construction is defined by the following rule

$$\begin{aligned} ackResponse(ack, inv, ok) \leftarrow\ & ack.method = \text{`}\texttt{ACK}\text{'} \\ \wedge\ & ack.to = ok.to \\ \wedge\ & ack.callId = inv.callId \\ \wedge\ & ack.from = inv.from \\ \wedge\ & ack.reqURI = inv.reqURI \\ \wedge\ & ack.cseq.seq = inv.cseq.seq \\ \wedge\ & ack.cseq.method = \text{`}\texttt{ACK}\text{'} \\ \wedge\ & ack.via.top = inv.via.top \end{aligned}$$

The following invariant, $I(\phi_2, \psi_2)$, serves to evaluate the property that every successful request should be acknowledged.

$$\begin{aligned} \phi_2 = \ & \forall_x (request(x) \wedge x.method = \text{`}\texttt{INVITE}\text{'} \rightarrow \\ & \quad \exists_{y>x} (responds(y, x) \wedge success(y) \rightarrow \exists_{z>y} ackResponse(z, x, y))) \\ \psi_2 = \ & \exists_{w>y} bye(y, w) \end{aligned}$$

where *success* accepts all success responses

$$success(x) \leftarrow 199 < x.statusCode \wedge x.statusCode < 300$$

and the failure criteria for the invariant is based on finding a session terminating request (`BYE`), which provides indication that the session was initiated without an `ACK` message appearing in the trace. The predicate *bye* is defined as

$$\begin{aligned} bye(ok, bye) \leftarrow\ & bye.method = \text{`}\texttt{BYE}\text{'} \\ \wedge\ & bye.callId = ok.callId \\ \wedge\ & bye.to = ok.to \\ \wedge\ & bye.from = ok.from \end{aligned}$$

The results of evaluation of the invariant are shown in Table 5.4. Since the premise of the test property (ϕ_2) is much more restrictive that for the first invariant, very few **pass** cases are reported by the tool and most result are vacuous (not shown in the table). This can be observed in traces 1, 5 and 8, where only vacuous results were reported, since these traces mostly contain `SUBSCRIBE` and `NOTIFY` messages, and no responses to `INVITE` messages appear.

In traces 3 and 6, a **fail** verdict was produced, meaning that a session was initiated without acknowledgment from the client, determined by the appearance of a session termination message `BYE` as indicated by the condition ψ_2. Since a single **fail** verdict was produced in comparison to several **pass**, the result may be indicative of an error in the collection of the trace, and it is not necessarily conclusive. Nevertheless it shows the effectiveness of our approach to detect inconsistent behavior.

Table 5.4: Results of testing the property "*every session initialization must be acknowledged*"

Trace	No. of messages	**pass**	**fail**	**inconclusive**	Time (s)
1	31	0	0	0	0.354
2	62	4	0	0	0.141
3	126	7	1	0	0.323
4	141	6	0	0	0.237
5	189	0	0	0	0.244
6	190	5	1	0	0.410
7	214	7	0	1	0.482
8	331	0	0	0	0.391
9	409	4	0	0	0.445
10	625	4	0	0	0.643

No session can be initiated without a previous registration

This property can be used to test that only users successfully registered with the SIP Core can initiate a PoC session (or a SIP call, depending on the service). The test part of the invariant is defined as follows

$$\begin{aligned}\phi_3 = \quad & \forall_x(\exists_{y>x} sessionEstablished(x,y) \\ & \rightarrow \exists_{u<x}(\exists_{v>u} registration(u,v)))\end{aligned}$$

where *sessionEstablished* and *registration* are defined as

$$\begin{aligned} sessionEstablished(x,y) \leftarrow \quad & x.method = \text{`}\texttt{INVITE}\text{'} \\ \wedge \quad & y.statusCode = 200 \\ \wedge \quad & responds(y,x) \end{aligned}$$

$$\begin{aligned} registration(x,y) \leftarrow \quad & request(x) \wedge responds(y,x) \\ \wedge \quad & x.method = \text{`}\texttt{REGISTER}\text{'} \\ \wedge \quad & y.statusCode = 200 \end{aligned}$$

The conditional part of the invariant is, however, not as easy to define in this case. The analysis of the results depends on the conditions of collections of the trace and whether the assumption that the trace is "*long enough*" holds. Unfortunately in the traces collected, such assumption could not be done, therefore a *false* condition

was used as in $I(\phi_3, \perp)$. Nevertheless it can be used to demonstrate the detection capabilities of the approach, as shown by the last row on Table 5.5. From the results it can be seen that traces 1, 5 and 8 produce only vacuous results. This is due to the fact that they mostly consist of `SUBSCRIBE` and `NOTIFY` messages, and therefore the condition *sessionEstablished*() never holds on the trace.

Table 5.5: Results of testing the property "*No session can be initiated without a previous registration*" on the set of traces.

Trace	No. of messages	**pass**	**fail**	**inconclusive**	Time (s)
1	31	0	0	0	0.744
2	62	0	0	4	1.13
3	126	0	0	8	2.726
4	141	0	0	6	1.869
5	189	0	0	0	1.714
6	190	0	0	6	2.851
7	214	0	0	8	4.494
8	331	0	0	0	4.588
9	409	0	0	4	10.155
10	625	4	0	0	38.874

From the results on Table 5.5, it can also be seen that the evaluation of this property is much more time consuming than the one on Table 5.3. Although this is expected given the complexity of evaluation described on Section 5.4.5.3 (n^2 from the first property vs. n^4 in the current one), the current definition of the property is also quite inefficient, and shows a possible limitation of the syntax. During evaluation, all combinations of x and y are tested until $sessionEstablished(x, y)$ becomes true, and then all combinations of u and v are evaluated until $registration(u, v)$ becomes true. It would be much more efficient to look first for a message with method `INVITE`, then look whether the invitation was validated by the server as a response with status `200` to then attempt to look for a registration. This could be achieved, for instance, by allowing quantifiers on the clause definitions, unfortunately, the syntax as currently specified does not allow that type of definition.

Subscription to events and notifications

As described in Chapter 3, in the presence service, a user (the watcher) can subscribe to another user's (the presentity) presence information, this works by using the SIP messages `SUBSCRIBE`, `PUBLISH` and `NOTIFY` for subscription, update and notification respectively. These messages also allow the subscription to other types of events other than presence, which is indicated in the header `Event` on the SIP message. It is desirable then to test, that whenever there is a subscription, a notification MUST occur upon an update event. This can be tested with the following formula for the

test part of an invariant $I(\phi_4, \bot)$.

$$\begin{aligned}\phi_4 = \quad & \forall_x(\exists_{y>x}(subscribe(x, watcher, user, event) \\ & \wedge update(y, user, event)) \\ & \rightarrow \exists_{z>y} notify(z, watcher, user, event))\end{aligned}$$

where *subscribe*, *update* and $notify$ hold on `SUBSCRIBE`, `PUBLISH` and `NOTIFY` events respectively. Notice that the values of the variables *watcher*, *user* and *event* may not have a value at the beginning of the evaluation, in that case their value is set by the evaluation of the *subscribe* clause, shown in the following

$$\begin{aligned}& subscribe(x, watcher, user, event) \\ \leftarrow \quad & x.method = \text{`SUBSCRIBE'} \\ \wedge \quad & watcher = x.from \\ \wedge \quad & user = x.to \\ \wedge \quad & event = x.event\end{aligned}$$

Here, the = operator, compares the two terms, however if one of the terms is an unassigned variable, then the operator works as an assignment. In the formula, the values assigned on the evaluation of *subscribe* will be then used for comparison in the evaluation of *update*. This is another way of defining formulas, different from just using messages as attributes.

The results of evaluating the formula are shown on Table 5.6. The results show no inconclusive results, although they also show that the full notification sequence is not present in most traces, with the exception of traces 9 and 10. Notice that we are explicitly looking for a sequence $subscribe \rightarrow update \rightarrow notify$, however the sequence $subscribe \rightarrow notify$ can also be present for subscription to server events, therefore `SUBSCRIBE` and `NOTIFY` events might also appear on the trace. To test the capabilities of detection, some `SUBSCRIBE` messages were manually introduced on a trace, matching existing `PUBLISH` messages. The lack of notification for the update was correctly detected as an **inconclusive** verdicts by the evaluation algorithm.

Similarly to property defined for registration, this property is quite inefficient in its evaluation, due to the same nesting of quantifiers. The evaluation time can be improved by rewriting the property as

$$\begin{aligned}& \forall_x(update(x, user, event) \\ & \quad \rightarrow (\exists_{y<x} subscribe(y, watcher, user, event) \\ & \quad\quad \rightarrow \exists_{z>x} notify(z, watcher, user, event)))\end{aligned}$$

which can be understood as: "*if an update event is found, then if a previous subscription exists to such event, a notification must be provided at some point after the update event*". The results of evaluating this property are shown on Table 5.7. Notice that for trace 9, a different number of true results are returned. This is due to the order of search given by the property, in the previous property finding one pair `SUBSCRIBE` – `PUBLISH` was enough to return a result. In the current property, for each `PUBLISH` it will look for a matching `SUBSCRIBE`. Since for every subscription there can exist multiple updates, the number of true results differs.

Table 5.6: Results of testing the property "*Whenever an update event happens, subscribed users must be notified*" on the set of traces.

Trace	No. of messages	**pass**	**fail**	**inconclusive**	Time (s)
1	31	0	0	0	0.59
2	62	0	0	0	0.677
3	126	0	0	0	1.157
4	141	0	0	0	0.84
5	189	0	0	0	1.618
6	190	0	0	0	1.418
7	214	0	0	0	1.602
8	331	0	0	0	3.994
9	409	3	0	0	6.033
10	625	4	0	0	14.972

Table 5.7: Results of testing the property "*Whenever an update event happens, subscribed users must be notified*" on the set of traces. Second version.

Trace	No. of messages	**pass**	**fail**	**inconclusive**	Time (s)
1	31	0	0	0	0.416
2	62	0	0	0	0.312
3	126	0	0	0	0.609
4	141	0	0	0	0.365
5	189	0	0	0	0.38
6	190	0	0	0	0.273
7	214	0	0	0	0.338
8	331	0	0	0	0.272
9	409	4	0	0	0.479
10	625	4	0	0	0.563

5.7 Comparison to related work

A number of different approaches to the testing and monitoring of formulas in traces exist in the literature for passive testing and runtime monitoring. In the following, we describe works in both categories in relation with their ability to express data relations for defining properties.

5.7.1 Passive testing

Although most works from passive testing that provide the basis of our approach have been previously cited [Lee 2002, Cavalli 2003, Arnedo 2003, Bayse 2005, Ladani 2005]. Some other works are worth mentioning with relation to our approach.

In some recent work, the authors of [Morales 2010] define a methodology for the definition and testing of time extended invariants, where data is also a fundamental

principle in the definition of formulas and a *packet* (similar to a *message* in our work) is the base container data. In this approach, the satisfaction of the packets to certain *events* is evaluated, and properties are expressed as $e_1 \xrightarrow{When,n,t} e_2$, where e_1 and e_2 are events defined as a set of constraints on the data fields of packets, n is the number of packets where the event e_2 should be expected to occur after finding e_1 in the trace, and t is the amount of time where event e_2 should be found on the trace after (or before) event e_1. This work served partly as inspiration for the work in this thesis. However, in our work we improve on it by allowing the definition of formulas that test data relations between multiple messages/packets.

Although closer to runtime monitoring, the authors of [Cao 2010] propose a framework for defining and testing security properties on Web Services using the Nomad [Cuppens 2005] language, based on previous works by the authors of [Li 2006, Li 2005]. As a work on web services, data passed to the operations of the service is taken into account for the definition of properties, and multiple events in the trace can be compared, allowing to define, for instance, properties such as "Operation *op* can only be called between operations *login* and *logout*". Nevertheless, in web services, operations are atomic, that is, the invocation of each operation can be clearly followed in the trace, which is not the case with network protocols, where operations depend on many messages and sometimes on the data associated with the messages.

5.7.2 Runtime monitoring

Runtime monitoring and runtime verification techniques have gained momentum in the latest years, particularly using model checking techniques for testing properties on the trace. The authors of [Leucker 2009] provide a good survey and introduction of methodologies in this area. The usual approach, consists on the definition of some logic (LTL is commonly used), which is used to create properties from which a *monitor* is defined to test on the trace. The authors of [Bauer 2007a] describe the definition of monitors as finite state machines for LTL formulas, they introduce a 3-valued semantics (true, false, inconclusive) in order to test formulas for finite segments of the trace[10], in [Bauer 2007b] they expand their analysis on inconclusive results, by proposing a 4-value semantics to distinguish cases where the property is most likely to become true or become false on the continuation of the trace. The analysis provided on this work on finite and infinite traces is based on the definitions from these authors, applied to our logic.

Regarding the inclusion of data, the concept of *parameterized propositions* is introduced by the authors of [Stolz 2008]. Propositions can contain data variables and quantifiers can be defined for the data variables by the introduction of a $\rightarrow$ operator, formulas of type $Q_1 x_1 \cdots Q_m x_m : p(x_1, \ldots, x_n) \rightarrow \psi$, where $Q_1, \ldots, Q_m$ are

[10] In their work, a trace segment is considered a finite word with an infinite continuation, so formulas that deal with the future of the trace have to take into account that the property can become true (or false) on the continuation of the trace.

quantifiers and $x_1, \ldots, x_m, \ldots, x_n$ are variables. In this approach, valid data values in formulas are fixed, so if $p(x)$ is used on the left side, the set $\{p(1), p(2), \ldots\}$ with the valid values must have been defined previously. Although it is an interesting approach to data testing, it is still propositional in nature, our approach adds flexibility to the definition of formulas by considering data as the central part of the communication.

A similar approach to ours is presented by the authors of [Roger 2001], for attack detection in logs. Their work uses a simplified LTL syntax where only the operators '$\wedge$' and '$\mathbf{F}$' are used. A formula $\{id = X, result =$ '`fail`'$\} \wedge \mathbf{F}\{id = X, result =$ '`pass`'$\}$ attempts to find a record in the log matching the first part (id and $result$ are fields in a record), and a future one matching the second part, where the variable X is assigned using a mechanism similar to unification. However, their approach focusing on attack detection, only deals with infinite traces, since definite verdicts are not a requirement.

Another work, defined to test message based work-flows, is provided by the authors of [Halle 2008] in the definition of the logic LTL-FO$^+$. Here, data is a more central part of the definition of formulas and LTL temporal operators are used to indicate temporal relations between messages in the trace. Messages are defined as a set of pairs $(label, value)$, similarly to our work, and formulas are defined with quantifiers specific to the labels. As an example, the formula $\mathbf{G}(\exists_{method} x_1 : x_1 =$ '`INVITE`' $\rightarrow \exists_{callId} x_2 : \mathbf{F}(\exists_{status} y_1 : y_1 = 200 \wedge \exists_{callId} y_2 : y_2 = x_2))$ indicates that generally, if a message with method `INVITE` is found, then it exists a field `Call-ID` in that message, such that a future message with status 200 exists with the same `Call-ID`. Although the syntax of the logic is flexible, it can quickly lose clarity as the number of variables required increases. Our current work improves on this, by allowing to group constraints with clause definitions.

Finally, in [Barringer 2004], the authors propose a logic for runtime monitoring of programs, called EAGLE, that uses the recursive relation from LTL $\mathbf{F}\phi \equiv \phi \vee \mathbf{X}\phi$ (and its analogous for the past), to define a logic based only on the operators *next* (represented by $\bigcirc$) and *previous* (represented by $\odot$). Formulas are defined recursively and can be used to define other formulas. Constraint on the data variables and time constraints can also be tested by their framework. However, their logic is propositional in nature and their representation of data is aimed at characterizing variables and variable expressions in programs, which makes it less than ideal for testing message exchanges in a network protocol as required in our work.

5.8 Conclusion

In this chapter we described our approach for data-centric invariant-based testing, motivated by the findings described in previous chapters related to the application of the invariant approach in modern protocol (and service) traces. There, we found that, when testing on traces for modern protocols (e.g. SIP), causality between

events in a trace can rarely be determined through control parts of the communication and it can only be determined through data parts.

In our work we define an invariant as a pair of formulas $I(\phi, \psi)$ where ϕ defines the *test* behavior and ψ defined the *conditional* behavior, i.e. the behavior that needs to be observed on the trace when evaluation of the test behavior does not produce sufficient information. A trace is considered in our work as a sequence of messages, i.e. collections of data fields. The syntax of formulas is defined in a bottom-up fashion: first, expected relations between messages and message data fields are defined as predicates using a syntax based on Horn clauses. Then the expected temporal behavior between messages and predicates is defined in order to test in the trace. The details of the syntax/semantics of formulas is defined in the chapter, along with an algorithm to determine satisfaction of a formula in a trace and an algorithm to provide a verdict of an invariant I, with respect to the behavior in the trace.

An example of definition of formulas for SIP-based services is provided, and the ability of the approach to detect failures in the trace is shown through experiments. The experiments also allowed to show some of the limitations of the work, namely the influence of the property definition in the evaluation time, and the difficulty of specifying conditional behavior to decide between *inconclusive* and *failed* verdicts. Some perspectives regarding future improvements and research paths are provided along with the general conclusion of our work in the next chapter.

Nevertheless, the expressibility and flexibility allowed by the used Horn-like definitions provides interesting perspectives for testing of protocol implementations and services.

Chapter 6

General Conclusion

The main objective of the presented work was to address some of the issues related to passive testing for conformance, particularly in the context of SIP-based services, IMS services and other message-based protocols.

We first presented, in Chapter 2, the state of the art of the most relevant works in passive testing, within the context of conformance testing with formal specifications. The definitions of the most commonly used models for passive testing were provided, and particular emphasis was given to the techniques of invariant-based passive testing, which are more relevant to our work. In that chapter, we also made a parallel with the concepts of runtime verification, given the common objective with invariant-based testing, of determining the satisfaction of a particular property in a trace. We noted that the general objectives of runtime verification differ with those of passive testing, however, many of the concepts and techniques can be used for the latter, and in fact have been used in the definition of our approach.

IMS services and SIP-based services, presented in Chapter 3, provide interesting challenges for conformance testing, and particularly for testing using traditional invariant-based testing approaches. In this context, several issues and limitations of the invariant approach are addressed by our work, some of them described as part of the work presented in Chapter 4. Invariant techniques are usually based on Finite State Machine (FSM) and Extended FSM models, and as such, they presume a causal relation between inputs and outputs (control parts) in the transitions of the model. In such context, properties (invariants) can be defined, as sequences of input/output pairs, that must be observed in the trace to determine conformance of the implementation. If causality between control parts is removed, e.g. when the implementation under test is the server in a centralized service, then causality can only be determined through data parts.

As traditional testing and verification approaches derive from finite state and labelled transition models, they center around control parts, with data parts defined as an extension, usually in the form of data fields or parameters. This means reduced expressiveness of invariant formulas to describe relations between multiple data fields, or relations more complex than equality. It also means reduced succinctness of formulas when adding data relations. In captured traces, however, events are usually messages, i.e. collections of structured data, and control parts are a function of the data. In the work presented in Chapter 5, we presented the details of our contribution: a message-based/data-centric approach for invariant testing.

In our work, to deal with expressiveness and succinctness issues, we define events in a trace as *messages*. Messages are defined as collections of data fields of different domain, where a particular data field value can be accessed through functions defined in our approach. Expected observations are defined as predicates on a message (or group of messages) in the form of Horn clauses. Each clause specifies a criteria to be fulfilled by a message or messages, defined in term of constraints over message data fields, for instance, that a particular data field of a message has a specific value or range of values, or that the field A of one message is equal to field B of another message. Temporal properties are then defined between different criteria, e.g. "if a message that satisfies criteria x is found, then a message satisfying criteria y must be found after it". The use of Horn clauses allows to define high-level criteria to be evaluated and re-used in multiple formulas. The fact that messages are evaluated, allows to define more general criteria than through control parts alone, for instance matching all requests or responses of a certain type.

Our approach defines temporal relations through quantification ($\forall$, $\exists$) over messages, with precedence of messages being specified through order relations. Then, a property can indicate that certain criteria "must be held for *all* messages in the trace", that exists "at least one message for which a particular criteria holds", or that for all messages that satisfy a given criteria, "another message must exist at some point in the future (after the position of the first message), satisfying a different criteria". This provides good temporal expressiveness, allowing to define temporal relations for the future and past of messages, or a mix of both. Since our approach has to deal with traces collected asynchronously, no immediate temporal precedence is defined (e.g. the *next* operator in LTL), given that it cannot generally be assured what the next message will be.

Dealing with temporal properties, when specifying that some event must occur at some point in the future (or past), another issue occurs. For a property such as "if event x happens, then event y must occur at some point in the future", if the event x is observed and y is not observed, distinguishing between the cases: "y was not produced" and "the trace collection ended before y" could be observed, is not trivial. The same issue occurs when dealing with events in the past, it cannot easily be determined if y was not produced or the trace collection started too early. In other invariant-based works, use of the specification was proposed to determine whether the initial state is present in the trace, however if the specification is not available, as it is many times the case, no solution is provided.

In our approach, such issue is directly considered in the design of the semantics of temporal formulas, where satisfaction of a formula can be determined within the set of truth values $\{\top, \bot, ?\}$, respectively indicating that the formula is satisfied, not satisfied and that no conclusive satisfaction result can be provided (inconclusive result). As evaluation of the behavior defined by a property "if x eventually y", is usually desired, '?' results are commonly obtained. Several alternatives have been proposed and implemented in our work for establishing which conformance verdict, **fail** or **inconclusive**, must be provided for a '?' satisfaction result.

1. Assume that the trace is never long enough, that is, only **inconclusive** verdicts can be provided for an '?' observation. Although this assumption is very strict, and may not provide useful information in most cases, analysis of the verdicts might. For instance, the number and distribution of **inconclusive** verdicts is significant, this may be an indication of a fault in the implementation.

2. Assume that the trace is always long enough, that is, assume that each '?' result is indication of a failure. If traces with large number of messages are used, this may be an acceptable alternative. However, some *false positive* may be produced at the edges of the trace.

3. Define an alternative, conditional, behavior to be observed. If while attempting to observe a given property, the conditional behavior is observed first, then a **fail** verdict is returned, otherwise, an **inconclusive** verdict is returned. This is similar to identifying the initial or final state of a specification in the trace for our more general case. Unfortunately, the condition does not always exist, or it may not be easy to define, since causality with the criteria in the expected property must be defined.

4. Explicitly define behavior that should not be observed, or negative behavior. Since the issue is establishing when the lack of observation of an expected behavior is indicative of a failure, there are no problems to detect when the behavior actually takes place. If a negative behavior is defined as a property, then, satisfaction of the property (a '$\top$' result) must produce a **fail** verdict. However, negative behavior may not always be easy to define from the requirements of the service.

Our definition of invariants includes all of these alternatives, an invariant can be *positive* or *negative*. A positive invariant, defined by $I(test, condition)$, where *test* and *condition* are temporal formulas defined with our message based syntax, evaluates the behavior defined by *test* in the trace, if the result is '?' and the behavior described by *condition* is observed, evaluation of the invariant provides a **fail** verdict. If the condition is '$\top$', a **fail** verdict is returned for each '?' result. If the condition is '$\bot$', an **inconclusive** verdict is returned for each '?' result. A *negative* invariant is defined by $I^{-}(test, condition)$, if the behavior described by *test* is observed, then a **fail** verdict is returned, otherwise, the condition is used to determine if a **pass** or **inconclusive** verdict should be returned for the evaluation of the invariant.

Our approach has been implemented into a prototype framework, and experiments for IMS service traces have been provided to exemplify how properties are defined with our message-based methodology, as well to illustrate the effectiveness of the approach to detect correct and incorrect behaviors.

6.1 Perspectives

In our work, we have provided a message-based approach for testing with conformance properties, which provides a novel work with respect to testing of data parts in other passive testing and runtime verification approaches. Considering trace events as messages, allows to define criteria for evaluation that include multiple control events, or even requirements over every observed message, no matter the control part. In the presented work, there is a direct relation between a message and its control part. It may also be possible to relate a criterion over a message with a set of control parts, e.g. the criterion 'all requests' with a subset of events in an EEFSM. This may provide useful, for instance, for verification of a property in a specification. However, it is also interesting to think about the possibility of defining message-based specifications, with transitions triggered by message reception and the satisfaction of a predicate over the message. To our knowledge, no such type of specification exists, and it may provide an alternative paradigm for testing. In the least, it may provide new directions for testing with message-based properties.

More particularly, in relation to improvements of our approach, as it can be seen from invariant definition and the results from the experiments in Section 5.6, inconclusive verdicts are many times unavoidable, and a conditional behavior for the invariant is not always possible to define. For a formula $\forall_x(p(x) \rightarrow \exists_{y>x} q(y,x))$, there is no way to decide when to stop the evaluation of the quantifier $\exists_{y>x} q(y,x)$, if no '$\top$' result for $q(y,x)$ is available, other than reaching the end of the trace. This issue and other possible improvements to the approach are discussed in the following.

6.1.1 Time constraints

A possible solution to distinguish between **fail** and **inconclusive** verdicts is to incorporate *time* constraints in the declaration of properties, whenever it might be required by the specification of the protocol. In this way, if the timeout indicated by a time constraint is reached before the end of the trace, a **fail** verdict must be returned, otherwise an **inconclusive** result is returned.

Extending the syntax of our logic to include time constraints is straightforward from the current definition. It would suffice to extend the possible operations inside trace quantifiers, for instance, something on the lines of $\forall_{y>x;y<x+t}\phi$ or $\forall_{y>x;y.time<x.time+t}\phi$ to indicate that the message to look for must occur after the current position of x and before the time given by the time-stamp of x plus t time units. Extending the semantics and evaluation algorithm should also be simple, the main restriction is that it must be able to distinguish between the end of the trace, in order to provide an inconclusive verdict, and the failure of a time constraint, to produce a false verdict.

6.1.2 Online evaluation

In many cases it would be desirable to allow a way to implement online testing of traces. That is, evaluate properties as the IUT is being run. This could be particularly interesting for testing security properties, or detecting clients with non-conforming implementations. In order to achieve this, three issues have to be dealt with in terms of the algorithm.

1. **Storage of traces to disk** Since our approach allows to define temporal quantifiers related to the past of a particular message, as well as the future, it is necessary that messages already collected and parsed can be accessed quickly and without the need to reprocessing them. This is actually a minor issue that can be solved, for instance, by the use of serialization on disk of the trace already collected. Nevertheless, the question of how far back to store the trace should be addressed, taking into account that the service may be run indefinitely and that the resources of the system are limited.

2. **Online evaluation of the condition**. Given an invariant $I(\phi, \psi)$, the current evaluation algorithm waits for a '?' result of the evaluation of ϕ, to test the conditional behavior. Since the evaluation of ϕ may run indefinitely, the conditional behavior must be evaluated along the test behavior, however, it cannot always be evaluated simultaneously. For instance, given an invariant $I(\forall_x(p(x) \rightarrow \exists_{y>x} q(y,x)), \exists_{z>x} r(z,x))$, the evaluation of the condition only makes sense if the evaluation of $p(x)$ yields a '$\top$' result, since other cases result in vacuous verdicts, and the evaluation of the condition is dependent on the value and position of x. The issue of *when* to start evaluating the condition must be considered.

3. **Parallel processing of the properties**. With the current evaluation algorithm, the evaluation of an invariant $I(\forall_x \phi, \psi)$ or $I(\exists_x \phi, \psi)$, runs indefinitely, until observing the conditional behavior, the end of the trace, or until finding a true evaluation for ϕ in the latter case. This raises a major issue for online testing, particularly in formulas with nested quantifiers. Let us take the test behavior defined by the formula $\forall_x \exists_{y>x} p(x,y)$ (and assuming that no conditional behavior exists). If for a given message in the trace, bound to x, no possible value for y exists such that $p(x,y)$ is true, then the evaluation will run forever without returning any further results, due to the fact that the trace collection may run indefinitely. A way to run evaluations in parallel, for instance, evaluating for a different binding for x without having to wait for a result for $\exists_{y>x} p(x,y)$ is necessary. A solution for this may be on the use of an alternating automata [Vardi 1995]. Nevertheless, the solution requires some study.

6.1.3 Improvements on the syntax/semantics

As seen in Section 5.6, and in accordance with the complexity analyzed in Section 5.4.5.3, whenever using nested quantifiers, the evaluation time increases proportionally to one order of magnitude with respect to the length of the trace for each nesting level. Given two equivalent properties, $\forall_x \exists_{y>x}(p(x) \wedge q(x,y))$ and $\forall_x(p(x) \wedge \exists_{y>x} q(x,y))$, processing time for the former is much larger than the latter, due to the fact that the first one tests every combination of values for x and y, and the latter only starts the search for a suitable value for y, after an x is found that makes $p(x)$ true.

Unfortunately, the current syntax and semantics does not allow to define formulas such as the one above, given that the separation between the evaluation of quantifiers and atomic formulas provided the simplest possible syntax to test our approach. The semantics of such a formula should follow directly from the current one using the definition of '$\wedge$', however the desired semantics for formulas of type '$? \wedge \phi$' has to be specified in order to produce a correct value. A partial solution can, however, be provided with the current approach, through the use of '$\rightarrow$'. Similar results can be obtained by a formula $\forall_x(p(x) \rightarrow \exists_{y>x} q(x,y))$, although the semantics differs.

A simple improvement can also be provided by allowing to define more complex relations between data fields, currently limited to '$=$', '$\neq$' and '$<$'. Addition of simple operations, for instance, can allow to test some more complex relations between data fields.

If a higher level definition of formulas is required, for added clarity and expressive power, it might be interesting to modify the syntax and semantics to allow quantification inside clause bodies. However, this requires a much more complex analysis and a complete redefinition of the semantics and definition of the evaluation procedure. Nevertheless it is an interesting improvement that is worth studying.

6.1.4 Race conditions in traces

Finally, an issue that also could merit more analysis is the fact that *race conditions* may occur on the trace, due to the distributed nature of the tested systems. With race conditions we refer to two or more messages of the protocol occurring at exactly the same time, i.e. appearing with the same timestamp on the trace. During the experiments some messages of this type were found although in this case they did not affect the evaluation of the properties. However, depending on the type of property and the testing purpose, it may occur that the same pair of messages causes a *false/inconclusive* or *true* verdict to be returned, depending on the order on which they are considered. This issue presents an interesting research challenge for the topic of passive testing.

Bibliography

[3rd Generation Partnership Project (3GPP) 2008] 3rd Generation Partnership Project (3GPP). *Internet Protocol (IP) multimedia call control protocol based on Session Initiation Protocol (SIP) and Session Description Protocol (SDP). Stage 3. TS 24.229*, March 2008. 25

[Abiteboul 1995] Serge Abiteboul, Richard Hull et Victor Vianu. Datalog and Recursion, chapitre 12, pages 271–310. Addison-Wesley, 1995. 66

[Ahson 2008] Syed Ahson et Mohammad Ilyas, editeurs. IP Multimedia Subsystem (IMS) Handbook. CRC Press, November 2008. 32

[Alcalde 2004] B. Alcalde, A. Cavalli, D. Chen, D. Khuu et D. Lee. *Network protocol system passive testing for fault management: A backward checking approach.* Lecture notes in computer science, pages 150–166, 2004. 18

[Apt 1982] K.R. Apt et M.H. Van Emden. *Contributions to the theory of logic programming.* Journal of the ACM (JACM), vol. 29, no. 3, pages 841–862, 1982. 74

[Apt 1997] K.R. Apt. From logic programming to Prolog, volume 368. Prentice Hall, 1997. 111

[Arnedo 2003] J. Arnedo, Ana Cavalli et M. Nunez. *Fast testing of critical properties through passive testing.* Testing of Communicating Systems, pages 608–608, 2003. 20, 21, 89

[Barringer 2004] Howard Barringer, Allen Goldberg, Klaus Havelund et Koushik Sen. *Rule-based runtime verification.* In Verification, Model Checking, and Abstract Interpretation, pages 277–306. Springer, 2004. 91

[Bauer 2006] Andreas Bauer, Martin Leucker et Christian Schallhart. *Monitoring of real-time properties.* In 26th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), pages 260–272. Springer-Verlag, 2006. vi, 4, 22, 70

[Bauer 2007a] Andreas Bauer et M Leucker. *Runtime verification for LTL and TLTL.* ACM Transactions on Software Engineering and Methodology, vol. X, pages 1–68, 2007. 23, 63, 90

[Bauer 2007b] Andreas Bauer, Martin Leucker et Christian Schallhart. *The good, the bad, and the ugly, but how ugly is ugly?* In Proceedings of the 7th international conference on Runtime verification, pages 126–138. Springer-Verlag, 2007. 90

[Bayse 2005] E. Bayse, A. Cavalli, M. Núñez et F. Zaïdi. *A passive testing approach based on invariants: application to the wap.* Computer Networks, vol. 48, no. 2, pages 247–266, 2005. vi, 4, 20, 41, 58, 59, 89

[Benharref 2007] Abdelghani Benharref, Rachida Dssouli, M.A. Serhani, A. En-Nouaary et Roch Glitho. *New Approach for EFSM-Based Passive Testing of Web Services.* Lecture Notes in Computer Science, vol. 4581, page 13, 2007. 18

[Bérard 2001] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci et P. Schnoebelen. Systems and software verification: model-checking techniques and tools. Springer, 2001. 23

[Bernot 1991] Gilles Bernot. *Testing against formal specifications: A theoretical view.* In TAPSOFT'91: International Joint Conference on Theory and Practice of Software Development, pages 1–18, Brighton, UK, 1991. Springer Verlag. 11

[Blum 2007] Niklas Blum, Fabricio Carvalho de Gouveia et Thomas Magedanz. *An Open IMS Testbed for exploring Wireless Service Evolution and Network Architecture Evolution towards SAE and LTE.* In The 2nd International Conference on Wireless Broadband and Ultra Wideband Communications (AusWireless 2007), pages 53–53. IEEE, August 2007. 40

[Bochmann 1994] Gregor V. Bochmann et Alexandre Petrenko. *Protocol testing: review of methods and relevance for software testing.* In Proceedings of the 1994 international symposium on Software testing and analysis - ISSTA '94, pages 109–124, New York, New York, USA, 1994. ACM Press. 12

[Bormann 2009] Matthias Bormann, Diederich Wermser et Ralf Patz. *Conformance Testing of Complex Services Exemplified with the IMS' Presence Service.* In 2009 Third International Conference on Next Generation Mobile Applications, Services and Technologies, pages 21–26. IEEE, September 2009. 40

[Brinksma 1988] E. Brinksma. *A theory for the derivation of tests.* In Protocol Specification, Testing, and Verification, volume 8, pages 63–74, 1988. 13

[Brinksma 1997] H. Brinksma, AW Heerink et GJ Tretmans. *Developments in testing transition systems.* In IFIP TC6 10th International Workshop on Testing of Communicating Systems, pages 143—-166, Kluwer, Cheju Island, Korea, 1997. Chapman & Hall. 14

[Brzezinski 2009] K.M. Brzezinski. Towards the methodological harmonization of passive testing across ict communities, pages 143–168. In-Tech, Oct 2009. 56

[Camarillo 2005] Gonzalo Camarillo et Miguel a. García-Martín. The 3G IP Multimedia Subsystem (IMS). John Wiley & Sons, Ltd, Chichester, UK, 3rd édition, December 2005. 32

[Camarillo 2008] G. Camarillo et A.B. Roach. *Framework and Security Considerations for Session Initiation Protocol (SIP) URI-List Services.* RFC 5363 (Proposed Standard), October 2008. 36

[Cao 2010] Tien-Dung Cao, Trung-Tien Phan-Quang, Patrick Felix et Richard Castanet. *Automated Runtime Verification for Web Services.* 2010 IEEE International Conference on Web Services, pages 76–82, July 2010. 90

[Cavalli 2003] Ana Cavalli, Svetlana Prokopenko et Caroline Gervy. *New approaches for passive testing using an Extended Finite State Machine specification.* Information and Software Technology, vol. 45, no. 12, pages 837–852, September 2003. 18, 19, 21, 89

[Chan 2003a] Ken Y. Chan et Gregor von Bochmann. *Methods for designing SIP services in SDL with fewer feature interactions.* In Proc. 7th. Feature Interactions in Telecommunications and Software Systems, pages 59—-76. IEEE, September 2003. 40

[Chan 2003b] Ken Y. Chan et Gregor von Bochmann. *Modeling IETF Session Initiation Protocol and its services in SDL.* In SDL Forum, pages 352–373. Springer, 2003. 40

[Chow 1978] T.S. Chow. *Testing Software Design Modeled by Finite-State Machines.* IEEE Transactions on Software Engineering, vol. SE-4, no. 3, pages 178–187, May 1978. 14

[Cuppens 2005] F. Cuppens, N. Cuppens-Boulahia et T. Sans. Nomad: A Security Model with Non Atomic Actions and Deadlines. IEEE, 2005. 90

[Day 2000] M. Day, J. Rosenberg et H. Sugano. *A Model for Presence and Instant Messaging.* RFC 2778 (Informational), February 2000. 35

[De Nicola 1984] R De Nicola. *Testing equivalences for processes.* Theoretical Computer Science, vol. 34, no. 1-2, pages 83–133, 1984. 12

[Delgado 2004] N. Delgado, A.Q. Gates et S. Roach. *A taxonomy and catalog of runtime software-fault monitoring tools.* IEEE Transactions on Software Engineering, vol. 30, no. 12, pages 859–872, December 2004. 22

[Dijkstra 1970] E.W. Dijkstra. *Notes on Structured Programming.* Technical report, Eindhoven Technological University, Eindhoven, Netherlands, 1970. 7

[ETSI/ES 201 873-1 2007] ETSI/ES 201 873-1. Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. V3.2.1. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, 2007. iii, 1, 9

[ETSI/ETR 022 1993] ETSI/ETR 022. *Advanced Testing Methods (ATM); Vocabulary of terms used in communication protocols conformance testing.* Technical report, 1993. iii, 1

[Falcone 2010a] Y. Falcone, J.C. Fernandez et Laurent Mounier. *What can you Verify and Enforce at Runtime?* Technical report, Verimag Research Report, 2010. 23

[Falcone 2010b] Yliès. Falcone, Jean-Claude. Fernandez, Thierry. Jéron et Hervé. Marchand. *More Testable Properties.* Technical Report April, Centre de recherche INRIA Rennes - Bretagne Atlantique, 2010. 23

[Fernandez 1991] J.C. Fernandez et Laurent Mounier. *A tool set for deciding behavioral equivalences.* In CONCUR '91. Proceedings of the 2nd International Conference on Concurrency Theory, pages 23–42. Springer, 1991. 12

[Frantzen 2004] L. Frantzen, J. Tretmans et T.A.C. Willemse. *Test generation based on symbolic specifications.* In FATES, volume 3395, pages 1–15. Springer, 2004. 13

[Fraser 2009] Gordon Fraser, Franz Wotawa et P.E. Ammann. *Testing with model checkers: a survey.* Software Testing, Verification and Reliability, vol. 19, no. 3, pages 215–261, 2009. 21, 79

[Garcia-Martin 2006] M. Garcia-Martin. *A Session Initiation Protocol (SIP) Event Package and Data Format for Various Settings in Support for the Push-to-Talk over Cellular (PoC) Service.* RFC 4354 (Informational), January 2006. 36, 40

[Gaudel 1995] M.C. Gaudel. *Testing can be formal, too.* In TAPSOFT'95: Theory and Practice of Software Development, volume 915, pages 82–96, Aarhus, Denmark, 1995. Springer Verlag. 11

[Giannakopoulou 2001] Dimitra Giannakopoulou et Klaus Havelund. *Runtime analysis of linear temporal logic specifications.* In Proceedings of the 16th IEEE International Conference on Automated Software Engineering, San Diego, California, numéro August, 2001. 22

[Halle 2008] Sylvain Halle et Roger Villemaire. *Runtime Monitoring of Message-Based Workflows with Data.* 2008 12th International IEEE Enterprise Distributed Object Computing Conference, pages 63–72, September 2008. 23, 63, 91

[Halle 2009] Sylvain Halle, Roger Villemaire et Omar Cherkaoui. *Specifying and Validating Data-Aware Temporal Web Service Properties.* IEEE Transactions on Software Engineering, vol. 35, no. 5, pages 669–683, September 2009. 23

[Handley 1999] M. Handley, H. Schulzrinne, E. Schooler et J. Rosenberg. *SIP: Session Initiation Protocol.* RFC 2543 (Proposed Standard), March 1999. Obsoleted by RFCs 3261, 3262, 3263, 3264, 3265. 25

[Havelund 2002] Klaus Havelund et Grigore Rosu. *Synthesizing Monitors for Safety Properties.* In Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 342–356, 2002. 22

[Havelund 2003] Klaus Havelund et Grigore Roşu. *Efficient monitoring of safety properties.* International Journal on Software Tools for Technology Transfer, vol. 6, no. 2, pages 158–173, November 2003. 22

[Hennine 1964] F. C. Hennine. *Fault detecting experiments for sequential circuits.* In 1964 Proceedings of the Fifth Annual Symposium on Switching Circuit Theory and Logical Design, pages 95–110. IEEE, November 1964. 14

[Hierons 2009] Robert M Hierons, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, Hussein Zedan, Kirill Bogdanov, Jonathan P Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman et Kalpesh Kapoor. *Using formal specifications to support testing.* ACM Computing Surveys, vol. 41, no. 2, pages 1–76, 2009. iii, 1, 9

[Hoffman 2008] Leah Hoffman. *In search of dependable design.* Communications of the ACM, vol. 51, no. 7, page 14, July 2008. 39

[IEEE Std 610.12-1990 1990] IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology. IEEE, New York, USA, 1990. 7

[ISO/IEC 9646 1994] ISO/IEC 9646. Information Technology – Open Systems Interconnection – Conformance testing methodology and framework. Geneva, Switzerland, 1994. iii, 1, 8

[ITU-T Z.100 1999] ITU-T Z.100. Recommendation Z.100 –Specification and Description Language. International Telecommunication Union (ITU-T), 1999. 9, 15

[Knuth 1977] Donald E. Knuth, James H. Morris Jr. et Vaughan R. Pratt. *Fast pattern matching in strings.* SIAM journal on computing, vol. 6, no. 2, pages 323–350, 1977. 41

[Krichen 2004] Moez Krichen et Stavros Tripakis. *Black-box conformance testing for real-time systems.* Model Checking Software, vol. 34, no. 3, pages 109–126, February 2004. 13

[Ladani 2005] B.T. Ladani, B. Alcalde et A. Cavalli. *Passive testing-a constrained invariant checking approach.* In Proc. 17th IFIP Int. Conf. on Testing of

Communicating Systems, pages 9–22. Springer, 2005. vi, 4, 21, 41, 53, 58, 89

[Lalanne 2009a] Felipe Lalanne et Stephane Maag. *From the IMS PoC service monitoring to its formal conformance testing.* In Proceedings of the 6th International Conference on Mobile Technology, Application & Systems - Mobility '09, pages 1–8, Nice, France, 2009. ACM Press. 40, 53

[Lalanne 2009b] Felipe Lalanne, Stephane Maag, Edgardo Montes De Oca, Ana Cavalli, Wissam Mallouli et Arnaud Gonguet. *An Automated Passive Testing Approach for the IMS PoC Service.* In 2009 IEEE/ACM International Conference on Automated Software Engineering, 2009. 40, 53

[Lalanne 2011a] Felipe Lalanne, Xiaoping Che et Stephane Maag. *Data-centric Property Formulation for Passive Testing of Communication Protocols.* In Applied Computing Conference. ACC'11, pages 176–181, Angers, France, 2011. 57

[Lalanne 2011b] Felipe Lalanne et Stephane Maag. *Protocol Data Parts Inclusion in a Formal Passive Testing Approach.* In 2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications, pages 569–573. IEEE, March 2011. 63

[Lee 1996] D. Lee et M. Yannakakis. *Principles and methods of testing finite state machines-a survey.* Proceedings of the IEEE, vol. 84, no. 8, pages 1090–1123, 1996. 9, 11, 14

[Lee 1997] D. Lee, A.N. Netravali, K.K. Sabnani, B. Sugla et A. John. *Passive testing and applications to network management.* In Proceedings 1997 International Conference on Network Protocols, pages 113–122. IEEE Comput. Soc, 1997. 14

[Lee 2002] D. Lee et Raymond E Miller. *A formal approach for passive testing of protocol data portions.* In 10th IEEE International Conference on Network Protocols, 2002. Proceedings., pages 122–131. IEEE Comput. Soc, 2002. 17, 89

[Lee 2006] D. Lee et R.E. Miller. *Network protocol system monitoring-a formal approach with passive testing.* IEEE/ACM Transactions on Networking, vol. 14, no. 2, pages 424–437, April 2006. 17

[Leucker 2009] Martin Leucker et Christian Schallhart. *A brief account of runtime verification.* Journal of Logic and Algebraic Programming, vol. 78, no. 5, pages 293–303, May 2009. iv, 2, 21, 56, 90

[Li 2005] Zheng Li, Jun Han et Yan Jin. *Pattern-based specification and validation of web services interaction properties.* In Service-Oriented Computing. ICSOC 2005, pages 73–86. Springer, 2005. 90

[Li 2006] Z. Li, Y. Jin et J. Han. *A runtime monitoring and validation framework for web service interactions.* In Software Engineering Conference, 2006. Australian, pages 10–pp. IEEE, 2006. 90

[Lloyd 1984] J.W. Lloyd. Foundations of logic programming. Springer-Verlag, 1984. 74

[Morales 2010] Gerardo Morales, Stephane Maag, Ana Cavalli, Wissam Mallouli et EM De Oca. *Timed Extended Invariants for the Passive Testing of Web Services.* In 8th IEEE International Conference of Web Services (ICWS 2010), 2010. 89

[Naito 1981] S. Naito et M. Tsunoyama. *Fault detection for sequential machines by transition tours.* In Proc. FTCS, volume 81, pages 238–243. IEEE Computer Society Press, 1981. 14

[Niemi 2004] A. Niemi. *Session Initiation Protocol (SIP) Extension for Event State Publication.* RFC 3903 (Proposed Standard), October 2004. 29

[Nilsson 1990] U. Nilsson et J. Maluszynski. Logic, programming and Prolog, volume 5. Wiley, 2nd édition, 1990. 68, 75

[Open Mobile Alliance 2006] Open Mobile Alliance. *Push to Talk over Cellular Requirements. Approved Version 1.0*, June 2006. v, 3, 36, 43

[Open Mobile Alliance 2009] Open Mobile Alliance. OMA PoC Control Plane – Approved Version 1.0.3. Open Mobile Alliance (OMA), 2009. 40, 41

[Open Mobile Alliance 2010] Open Mobile Alliance. *Presence SIMPLE. Candidate Enabler Release V2.0*, February 2010. v, 3, 35

[Panwar 2007] Birender Panwar et Keval Singh. *IMS SIP core server test bed.* In 2007 International Conference on IP Multimedia Subsystem Architecture and Applications, pages 1–5. IEEE, December 2007. 40

[Roach 2002] A. B. Roach. *Session Initiation Protocol (SIP)-Specific Event Notification.* RFC 3265 (Proposed Standard), June 2002. Updated by RFCs 5367, 5727. 29, 35, 36, 40, 51

[Roger 2001] M. Roger, M. Roger et J. Goubault-Larrecq. *Log auditing through model-checking.* In Proceedings from the 14th IEEE Computer Security Foundations Workshop (CSFW'01), numéro June, pages 220–236. IEEE Computer Society Press, 2001. 91

[Rosenberg 2002] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley et E. Schooler. *SIP: Session Initiation Protocol.* RFC 3261 (Proposed Standard), June 2002. Updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621, 5626, 5630, 5922, 5954, 6026, 6141. 3, 25, 33, 71, 84

[Rosenberg 2004] J. Rosenberg. *A Presence Event Package for the Session Initiation Protocol (SIP).* RFC 3856 (Proposed Standard), August 2004. 35

[Rosenberg 2006] J. Rosenberg, H. Schulzrinne et O. Levin. *A Session Initiation Protocol (SIP) Event Package for Conference State.* RFC 4575 (Proposed Standard), August 2006. 40, 51, 53

[Stolz 2008] V. Stolz. *Temporal Assertions with Parametrized Propositions.* Journal of Logic and Computation, vol. 20, no. 3, pages 743–757, November 2008. 22, 63, 90

[Sugano 2004] H. Sugano, S. Fujimoto, G. Klyne, A. Bateman, W. Carr et J. Peterson. *Presence Information Data Format (PIDF).* RFC 3863 (Proposed Standard), August 2004. 35

[Tabourier 1999] Marine Tabourier et Ana Cavalli. *Passive testing and application to the GSM-MAP protocol.* Information and Software Technology, vol. 41, no. 11-12, pages 813–821, September 1999. 16

[Tretmans 1992] Jan Tretmans. *A formal approach to conformance testing.* PhD thesis, University of Twente, 1992. 12

[Tretmans 1996a] J Tretmans. *Conformance testing with labelled transition systems: Implementation relations and test generation.* Computer Networks and ISDN Systems, vol. 29, no. 1, pages 49–79, December 1996. 14

[Tretmans 1996b] Jan Tretmans. *Test generation with inputs, outputs, and quiescence.* In Germany Passau, editeur, Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, pages 127–146. Springer, 1996. 13

[Tretmans 1999] Jan Tretmans. *Testing Concurrent Systems : A Formal Approach.* In 10th International Conference on Concurrency Theory, pages 46–65, Eindhoven, The Netherlands, 1999. Springer-Verlag. 10

[Tretmans 2001] Jan Tretmans. *An Overview of OSI Conformance Testing.* Technical report, University of Twente, Twente, The Netherlands, 2001. 8

[Tsagkaropoulos 2007] Michail Tsagkaropoulos, Ilias Politis et Tasos Dagiuklas. *IMS Evolution and IMS Test-Bed Service Platforms.* 2007 IEEE 18th International Symposium on Personal, Indoor and Mobile Radio Communications, pages 1–6, September 2007. 39

[Ural 1992] H. Ural. *Formal methods for test sequence generation.* Computer Communications, vol. 15, no. 5, pages 311–325, June 1992. 14

[Van Emden 1976] MH Van Emden et RA Kowalski. *The semantics of predicate logic as a programming language.* Journal of the ACM (JACM), vol. 23, no. 4, pages 733–742, 1976. 68, 70

[Vardi 1995] M. Vardi. *Alternating automata and program verification.* Computer Science Today, pages 471–485, 1995. 97

[Vardi 1996] M. Vardi. *An automata-theoretic approach to linear temporal logic.* In Proceedings of the VIII Banff Higher order workshop conference on Logics for concurrency structure versus automata, pages 238–266. Springer-Verlag, 1996. 22

[Woodcock 2009] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui et John Fitzgerald. *Formal methods: Practice and experience.* ACM Computing Surveys, vol. 41, no. 4, pages 1–36, October 2009. iii, 1

Appendix A

A Framework for Data-centric evaluation

The concepts and algorithms described in Chapter 5 have been implemented into a framework, available in the URL `http://www-public.int-evry.fr/~lalanne/thesis.html`, of which some details are provided in the following. The framework has been implemented in Java, and the general architecture is based on that of the IRIS reasoner project[1] , a reasoning engine for Datalog rules. However, most of the functionality of the different modules has been rebuilt to support compound domains and message selector variables, along the rest of the functionality (quantified formulas, SLD-resolution, etc.). Three main modules can be identified in the system: *i.* filtering and conversion of collected traces, *ii.* evaluation of invariants and *iii.* evaluation of formulas. Figure A.1 shows the module interaction and the inputs/outputs from each one, and a brief description of each module and its configuration is described in the following sections.

A.1 Trace processing module

The trace processing module receives the raw traces collected from network exchange and converts the messages from the input format to a list of messages, used by the formula evaluation module. Although the module is designed to be adaptable for any input format, in our particular implementation, the used format is PDML, an XML format that can be obtained from Wireshark[2] traces. The structure of the PDML is provided in figure A.2. In the XML, data fields are identified by a `field` tag, and are grouped by protocol. Each field tag represents an individual data value in a message (a header, a header sub-element). The field is identified by a `name` attribute and its value is hex encoded in the attribute `value`. Field tags can be nested to indicate complex data types. This nested structure corresponds quite closely with the message structure, where data can be grouped in compound domains.

The configuration for the trace conversion is shown in Figure A.3. Each identifier on the left side of a ':' symbol, indicates a sub-element of a message, and identifiers on the right, represent a field name in the XML. Notice that identifiers on the left side can represent compound domains, for instance, the configuration in the figure

[1] `http://www.iris-reasoner.org/`
[2] `http://www.wireshark.org`

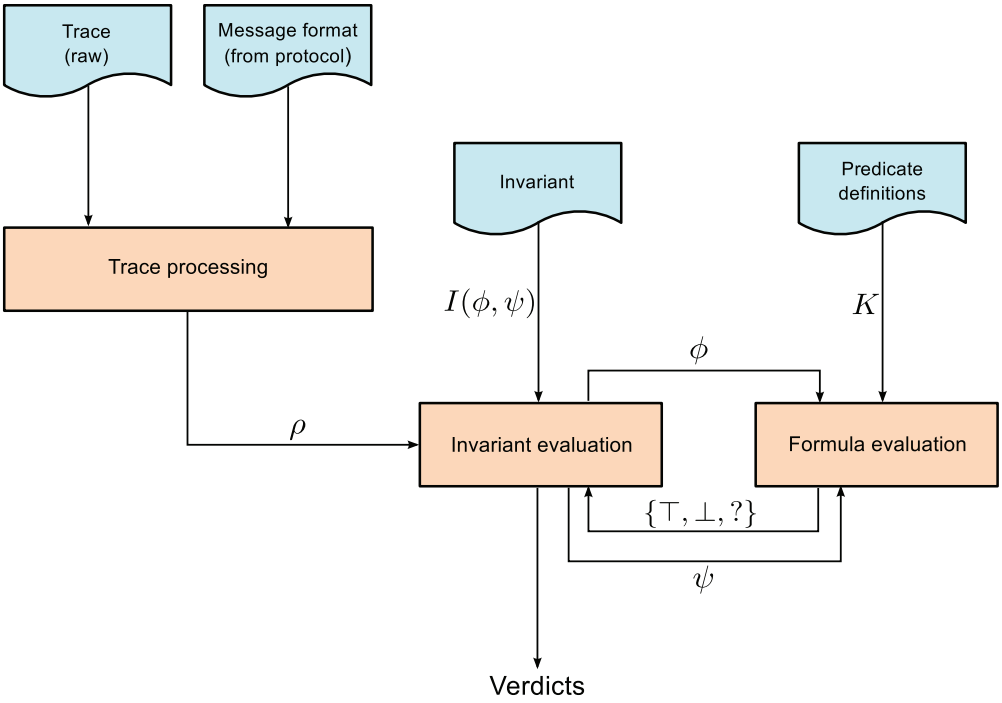


Figure A.1: Architecture for the framework.

```
<packet>
    <proto name="sip">
        <field name="sip.Method" show="INVITE" value="494e56495445"/>
        <field name="sip.From" show="alice@domain.org;tag=123"
            value="616c69636540646f6d61696e2e6f72673b7461673d313233">
            <field name="sip.from.addr" show="alice@domain.org"
                value="616c69636540646f6d61696e2e6f7267" />
            <field name="sip.to.tag" show="123" value="313233" />
        </field>
        ...
    </proto>
    <proto>...</proto>
</packet>
```

Figure A.2: Example of a PDML file structure

indicates that `status` is a sub-element of message, with two sub-values `status.line` and `status.code`. Although not included in the formal description of the approach, the framework supports list elements as well. In the figure, the field `via` defines a list, where each element of the list is a compound value, with sub-elements `transport`, `address`, `port` and `branch`. Elements of the list are identified in the PDML as fields with identical name. The configuration also allows to provide a main protocol under test. If a packet in the trace does not contain headers of the protocol indicated by the identifier `PROTOCOL`, then the packet is ignored.

```
define PROTOCOL "sip"

requestURI  : sip.r-uri
method      : sip.Method
status.line : sip.Status-Line
status.code : sip.Status-Code
to.addr     : sip.to.addr
to.tag      : sip.to.tag
from.addr   : sip.from.addr
from.tag    : sip.from.tag
cSeq.seq    : sip.CSeq.seq
cSeq.method : sip.CSeq.method

via         :
{
    transport   : sip.Via.transport
    address     : sip.Via.sent-by.address
    port        : sip.Via.sent-by.port
    branch      : sip.Via.branch
}
```

Figure A.3: Example of message configuration

A.2 Formula evaluation module

The formula evaluation module is called during invariant evaluation, although it can also be used independently. The module receives a trace (the result from the processing module), a set of predicate definitions, a formula to test, and an optional set of initial bindings (a substitution). It returns a result for each evaluation of the formula in the trace, returning the satisfaction result ($\{\top, \bot, ?\}$) and the variable bindings at the end of the evaluation.

Predicate definitions are provided as an input file to the tool. Clauses are defined as described in the Section 5.4.3.1 and the syntax for definition of clauses is similar to that of Prolog [Apt 1997], with the exception that variables are not required to be written in upper case, and strings have to be explicitly indicated between quotes. For instance, the clauses in the Example 5.4.4, are defined as

```
request(x)  :- x.method != nil.
response(x) :- x.statusCode != nil.
responds(x,y) :-
    response(x),
    x.from = y.from,
    comparable(y.to, x.to),
    x.callId = y.callId,
    x.cSeq = y.cSeq,
    x.via = y.via.
```

Notice that the definition of each clause is terminated by a '.', that the '$\wedge$' symbol is represented by a ',', equivalently to PROLOG, and ε is represented by '`nil`'. For the definition of queries, the following conventions are used

- The symbol $\rightarrow$ is represented by '`->`'.
- $\forall_x \phi$, $\forall_{y>x} \phi$, $\exists_x \phi$, $\exists_{y>x} \phi$, are respectively represented as `forall(x, ` ϕ`)`, `forall(y > x, ` ϕ`)`, `exists(x, ` ϕ`)` and `exists(y > x, ` ϕ`)`

The query $\forall_x(request(x) \rightarrow \exists_{y>x} responds(y, x))$ is then written as:

```
forall(x, request(x) -> exists(y > x, responds(y, x)))
```

A.3 Invariant evaluation module

The invariant evaluation module works in accordance to the procedure defined in Section 5.5. The module takes the trace from the trace processing module, as well as the definition of the invariant from the configuration files. The module gives first the test part of the invariant to the formula evaluation module and, for each result, it produces a verdict, **pass**, **fail** or **inconclusive**. If a condition is defined, then the module uses the bindings from the evaluation of the test to evaluate the condition. An invariant is defined in the configuration as a triple `[positive, test, condition]`, where positive is one of `+` or `-` and indicates whether the invariant is positive or negative. The condition can be a formula, or one of the values `true` or `false`. An example is provided as follows.

```
define INVARIANT [ "+",
    "forall(x, request(x) -> exists(y > x, responds(y, x))).",
    "false" ]
```

In Appendix B, the full configuration files for the experiments are provided.

Appendix B

Framework Configuration for Experiments

The configuration files for the framework, message configuration, predicate definitions and invariants used in Section 5.6 are provided in the following sections.

B.1 Tool and Message Configuration

```
/* The main protocol under test,
 * filters any message outside of this protocol */
define PROTOCOL 'sip'

/* Predicate definition files */
define PREDICATES 'sip.kb'

/* List of traces to evaluate */
define TRACE ['traces/patched/E2EIMSSPTT1003.xml',
'traces/patched/E2EIMSSPTT1001.xml',
'traces/patched/E2EIMSSPTT1004.xml',
'traces/patched/E2EIMSSPTW1001.xml',
'traces/patched/reg-WM8.0.4.5.xml',
'traces/patched/E2EIMSSPTT1017.xml',
'traces/patched/E2EIMSSPTT1005.xml',
'traces/patched/E2EIMSSPTT1018.xml',
'traces/patched/E2EIMSSPTT1019.xml',
'traces/patched/E2EIMSSPTT1009.xml'
]

/**
 * An invariant is a list with three elements.
 * - The first a "+" or "-" sign,
 *   indicating whether the invariant is positive or negative
 * - The second is the test part of the invariant
 * - The third one is the condition part of the invariant,
 */
```

```
define INVARIANT [ "+",
"forall(x, request(x), x.method != 'ACK' ->
        exists(y > x, non_provisional(y), responds(y,x))).",
"false" ]


/**
 * Message configuration.
 *
 * Each line defines a data field in the message (left) and
 * its equivalent field in the PDML (right). Notice that
 * defining a sub-field (status.line), defines the parent (status).
 *
 * Lists are also supported. An example is provided with the
 * 'via' field.
 */
requestURI  : sip.r-uri
method      : sip.Method
status.line : sip.Status-Line
status.code : sip.Status-Code
to.addr     : sip.to.addr
to.tag      : sip.to.tag
from.addr   : sip.from.addr
from.tag    : sip.from.tag
cSeq.seq    : sip.CSeq.seq
cSeq.method : sip.CSeq.method
callId      : sip.Call-ID
maxForwards : sip.Max-Forwards

via         : /* This indicates that via is a list */
{
    transport   : sip.Via.transport
    address     : sip.Via.sent-by.address
    port        : sip.Via.sent-by.port
    branch      : sip.Via.branch
}
```

B.2 SIP Predicate Definitions

```
/* Valid    Status codes */

/* Provisional Responses */
status(100). /* Trying */
```

```
status(180). /* Ringing */
status(181). /* Call is being forwarded */
status(182). /* Queued */
status(183). /* Session progress */
/* Success */
status(200). /* OK */
/* Redirection */
status(300). /* Multiple Choices */
status(301). /* Moved permanently */
status(302). /* Moved temporarily */
status(305). /* Use Proxy */
status(380). /* Alternative service */
/* Client errors */
status(400). /* Bad request */
status(401). /* Unauthorized */
status(402). /* Payment required */
status(403). /* Forbidden */
status(404). /* Not found */
status(405). /* Method not allowed */
status(406). /* Not acceptable */
status(407). /* Proxy Authentication Required */
status(408). /* Request timeout */
status(410). /* Gone */
status(413). /* Request entity too large */
status(414). /* Request-URI Too Large */
status(415). /* Unsupported Media Type */
status(416). /* Unsupported URI Scheme */
status(420). /* Bad extension */
status(421). /* Extension required */
status(423). /* Interval too brief */
status(480). /* Temporarily not available */
status(481). /* Call Leg/Transaction Does Not Exist */
status(482). /* Loop detected */
status(483). /* Too Many Hops */
status(484). /* Address Incomplete */
status(485). /* Ambiguous */
status(486). /* Busy Here */
status(487). /* Request terminated */
status(488). /* Not Acceptable Here */
status(489). /* Bad Event */
status(491). /* Request pending */
status(493). /* Undecipherable */
/* Server errors */
status(500). /* Internal Server Error */
```

```
status(501). /* Not implemented */
status(502). /* Bad Gateway */
status(503). /* Service Unavailable */
status(504). /* Server Time-out */
status(505). /* SIP Version not supported */
status(513). /* Message Too Large */
/* Global-Failure */
status(600). /* Busy everywhere */
status(603). /* Decline */
status(604). /* Does not exist anywhere */
status(606). /* Not acceptable */


/* RFC 3621 methods */
method('REGISTER').
method('INVITE').
method('ACK').
method('CANCEL').
method('BYE').
method('OPTIONS').

/* Other methods (from presence) */
method('PUBLISH').
method('NOTIFY').
method('SUBSCRIBE').


/* Response types */

/* Provisional responses 1xx */
provisional(x) :-
    x.status.code >= 100,
    x.status.code < 200.

/* Success reponses: 2xx */
success(x) :-
    x.status.code >= 200,
    x.status.code < 300.


/* Redirections: 3xx */
redirection(x) :-
    x.status.code >= 300,
    x.status.code < 400.
```

```
/* Client errors: 4xx */
client_error(x) :-
    x.status.code >= 400,
    x.status.code < 500.

/* Server errors: 5xx */
server_error(x) :-
    x.status.code >= 500,
    x.status.code < 600.

/* Global failures: 6xx */
global_failure(x) :-
    x.status.code >= 600,
    x.status.code < 700.

/* True if the message is a request */
request(x) :- method(x.method).

/* True if the message is a response */
response(x) :- status(x.status.code).

/* A non-provisional response */
non_provisional(x) :-
    x.status.code >= 200,
    x.status.code < 700.

/* True if message x is a response to message y.
 * Assumes y is a request message */
responds(x,y) :-
    response(x),
    x.from = y.from,
    compareTo(y.to, x.to),
    x.callId = y.callId,
    x.cSeq = y.cSeq,
    x.via = y.via.

/* Compares the to fields from the request
   and the response */
compareTo(reqTo, respTo) :- reqTo.tag = nil,
    reqTo.addr = respTo.addr.
compareTo(reqTo, respTo) :- reqTo.tag = respTo.tag,
    reqTo.addr = respTo.addr.
```

```
/* Session established */
sessionEstablished(x,y) :-
    x.method = 'INVITE',
    y.status.code = 200,
    responds(y,x).


/* Subscription */
subscribe(x, watcher, user, event) :-
    x.method = 'SUBSCRIBE',
    watcher = x.from,
    user = x.to,
    event = x.event.


/* Update status */
update(x,user,event) :-
    x.method = 'PUBLISH',
    x.from = user,
    x.event = event.

/* Notify */
notify(x, watcher, user, event) :-
    x.method = 'NOTIFY',
    x.from = user,
    x.to = watcher,
    x.event = event.

/* Registration */
registration(x,y) :- request(x), responds(y,x),
    x.method = 'REGISTER', y.status.code = 200.

ackResponse(ack,inv,ok) :-
    ack.method = 'ACK',
    ack.callId = inv.callId,
    ack.cSeq.seq = inv.cSeq.seq,
    ack.cSeq.method = 'ACK',
    ack.to = ok.to,
    ack.from = inv.from,
    head(ackTop, ack.via),
    head(invTop, inv.via),
    ackTop.address = invTop.address,
    ackTop.port = invTop.port,
```

```
    ackTop.transport = invTop.transport.

bye(ok, bye) :-
    bye.method = 'BYE',
    bye.callId = ok.callId,
    bye.to = ok.to,
    bye.from = ok.from,
    bye.route = ok.recordRoute.
```

B.3 Invariant definitions

For every request there must be a response

```
define INVARIANT [ "+",
    "forall(x, request(x), x.method != 'ACK' ->
        exists(y > x, non_provisional(y), responds(y,x))).",
    "false" ]
```

Every session initialization must be acknowledged

```
define INVARIANT [ "+",
    "forall(x, request(x), x.method = 'INVITE' ->
         (exists(y > x, responds(y,x), success(y)) ->
                exists(z > y, ackResponse(z,x,y)))
        ).",
    "exists(w > y, bye(y,w))." ]
```

No session can be initiated without a previous registration

```
define INVARIANT ["+",
    "forall(x, exists(y > x, sessionEstablished(x,y)) ->
        exists(u < x, exists(v > u, registration(u,v)))).",
    "false"]
```

Subscription to events and notifications

The first version of this property is defined as follows.

```
define INVARIANT ["+",
    "forall(x, exists(y > x, subscribe(x,watcher,user,event),
        update(y,user,event)) ->
            exists(z > y, notify(z,watcher,user,event))).",
    "false"]
```

The improved version is defined by the following definition.

```
define INVARIANT ["+",
    "forall(x, update(x, user, event) ->
        (exists(y < x, subscribe(y, watcher, user, event)) ->
             exists(z > x, notify(z, watcher,user,event)))).",
    "false"]
```

Résumé : Le test de conformité est le processus permettant de contrôler qu'un système possède un ensemble de propriétés souhaitées et se comporte conformément à certaines exigences prédéfinies. Dans ce contexte, les techniques de *test passif* sont utilisées lorsque le système sous test ne peut être interrompu ou l'accès aux interfaces du système est indisponible. Le test passif s'appuie sur l'observation de l'application pendant l'exécution, et la comparaison de l'observation avec le comportement attendu, défini à travers des propriétés de conformité.
L'objectif de cette thèse est la définition d'une méthodologie de validation des protocoles communicants par test passif. Les approches existantes sont issues de travaux basés sur des spécifications à états finis ou de transitions étiquetées et comme tels, ils présument l'existence d'une relation de causalité entre les évènements observés dans la trace du système. Pour le traitement des protocoles basés sur des messages, comme le protocole SIP (fondamental pour les services IMS), telle causalité n'existe pas nécessairement et en outre, elle ne peut être déterminée que par la partie données du protocole. Étant donné que les techniques existantes sont optimisées pour traiter les parties de contrôle, ils présentent des limites pour les tests basés sur des parties de données: expressibilité réduite de propriétés de conformité, entre autres.
Dans ce travail nous présentons une approche sur la base des messages et données pour traiter ces problèmes. Les observations dans une trace sont sous la forme de messages. Le comportement attendu est défini de manière ascendante, à partir des critères basés sur les relations entre les champs des données des messages. Des relations temporelles sont définies entre ces critères, par exemple, une propriété peut exiger que certains critères "doit être reconnu pour *tous* les messages dans la trace". Notre approche permet d'exprimer des formules sur l'avenir et le passé de la trace, permettant de définir des critères plus généraux que ceux qui utilisent uniquement des parties de contrôle.
Des problèmes liés à la satisfaction des propriétés et la déclaration des verdicts de conformité sont également discutés. Bien que l'observation d'un comportement défini comme une propriété est un indice de conformité, l'absence d'observation n'est pas nécessairement indicative d'une faute. Plusieurs solutions à ce problème ont été proposées et mises en œuvre dans ce travail.
Enfin, notre travail présente des perspectives intéressantes en termes d'extensibilité pour la détection en ligne ou une expressivité améliorée, mais aussi car une approche basée sur des messages fournit une vision alternative aux techniques de test traditionnelles.

Mots clés : Test, conformité, passif, données, IMS, protocoles, services

Modeling and Methodologies for the Test of IMS Services

Abstract: Conformance testing is the process of checking that a system possesses a set of desired properties and behaves in accordance with some predefined requirements. In this context, *passive* testing techniques are used when the system under test cannot be interrupted or access to the system's interfaces is unavailable. Passive testing relies on the observation of the implementation during runtime, and the comparison of the observation with the expected behavior, defined through conformance properties.

The objective of this thesis is to define a novel methodology to validate communicating protocols by passive testing. Existing approaches are derived from works with finite-state and labelled transition specifications and as such, they presume there exists a causality relation between the events observed in the implementation (the trace). When dealing with message-based protocols, such as the Session Initiation Protocol (fundamental for IMS services), such causality does not necessarily exist and furthermore, it may only be determined through data parts. Since existing techniques are optimized for dealing with control parts, they present limitations for testing based on data parts: reduced expressibility and succinctness of conformance properties, as well as problems to deal with satisfaction of properties including future conditions.

In this work we present a message-based/data-centric approach for dealing with these issues. Observations in a trace are in the form of *messages*. Expected behavior is defined in a bottom-up fashion, starting from expected criteria that must be fulfilled by one or more messages, defined as constraints between the message data fields. Temporal relations by quantification over the criteria, e.g. a property may require that certain criteria "must be held for *all* messages in the trace". Our approach allows to express formulas about the future and past of the trace, allowing to define more general criteria than through control parts alone.

Issues related to satisfaction of properties and declaration of conformance verdicts are also discussed here. Although observation of a behavior defined as a property is indication of conformance, lack of observation is not necessarily indicative of a fault. Several solutions to this issue have been proposed and implemented in this work.

Finally, our work presents interesting perspectives, in terms of extensibility for online detection or improved expressiveness, but also since a message-based approach provides an alternative view to traditional testing techniques.

Keywords: Testing, conformance, passive, data, IMS, protocols, services