A formal model for accountability

Walid Benghabrit

Thèse de Doctorat

Walid BENGHABRIT

Mémoire présenté en vue de l'obtention du

grade de Docteur de l'école nationale supérieure Mines-Télécom Atlantique Bretagne-Pays de la Loire - IMT Atlantique

sous le sceau de l'Université Bretagne Loire

École doctorale : *Mathématiques et STIC (MathSTIC)*

Discipline : *Informatique, section CNU27*

Spécialité : *Sécurité et sûreté de l'information*

Unité de recherche : *Le Laboratoire des Sciences du Numérique de Nantes (LS2N)*

Soutenue le *: 27/10/2017*

Thèse N° : 2017IMTA0043

UN MODEL FORMEL POUR LA RESPONSABILISATION

« A FORMAL MODEL FOR ACCOUNTABILITY »

JURY

Rapporteurs :	**Mme. Ana Rosa CAVALLI**, Professeur des Universités, Telecom SudParis
	M. Yves ROUDIER, Professeur des Universités, Université de Nice Sophia Antipolis
Examinateurs :	**M. Daniel LE METAYER,** Directeur de Recherche, Inria
	Mme. Nora CUPPENS, Directrice de Recherche, IMT ATLANTIQUE
	M. Anderson SANTANA DE OLIVEIRA, Chercheur Sénior, SAP Labs
Directeur de Thèse :	**M. Jean-Claude ROYER**, Professeur des Mines, IMT ATLANTIQUE
Co-encadrant de Thèse :	**M. Hervé GRALL**, Maître de conférences, IMT ATLANTIQUE

A FORMAL MODEL FOR ACCOUNTABILITY

WALID BENGHABRIT

-

Computer Science
IMT Atlantique
ASCOLA Research Group
Supervisor : Pr. Jean-Claude Royer
Co-supervisor : A/Prof. Hervé Grall

– OCT 2017 –

"By the time I realized my parents were right, I had kids that didn't believe me."

— **Hussein Nishah**

Dedicated to my parents.

ABSTRACT

Nowadays we are witnessing the democratization of cloud services. As a result, more and more end-users (individuals and businesses) are using these services in their daily lives. In such scenarios, personal data is generally flowed between several entities. End-users need to be aware of the management, processing, storage and retention of personal data, and to have necessary means to hold service providers accountable for the use of their data. In this thesis we present an accountability framework called Accountability Laboratory (AccLab) that allows to consider accountability from design time to implementation time of a system. In order to reconcile the legal world and the computer science world, we developed a language called Abstract Accountability Language (AAL) that allows to write obligations and accountability policies. This language is based on a formal logic called First Order Linear Temporal Logic (FOTL) which allows to check the coherence of the accountability policies and the compliance between two policies. These policies are translated into a temporal logic called $FO\text{-}DTL^3$, which is associated with a monitoring technique based on formula rewriting. Finally, we developed a monitoring tool called Accountability Monitoring (AccMon) which provides means to monitor accountability policies in the context of a real system. These policies are based on $FO\text{-}DTL^3$ logic and the framework can act in both centralized or distributed modes and can run in on-line and off-line modes.

RÉSUMÉ

Nous assistons à la démocratisation des services du cloud et de plus en plus d'utilisateurs (individuels ou entreprises) utilisent ces services dans la vie de tous les jours. Dans ces scénarios les données personnelles transitent généralement entre plusieurs entités. L'utilisateur final se doit d'être informé de la collecte, du traitement et de la rétention de ses données personnelles, mais il doit aussi pouvoir tenir pour responsable le fournisseur de service en cas d'atteinte à sa vie privée. La responsabilisation (ou accountability) désigne le fait qu'un système ou une personne est responsable de ses actes et de leurs conséquences.

Dans cette thèse nous présentons un framework de responsabilisation AccLab qui permet de prendre en considération la responsabilisation dès la phase de conception d'un système jusqu'à son implémentation. Afin de réconcilier le monde juridique et le monde informatique, nous avons développé un langage dédié nommé AAL permettant d'écrire des obligations et des politiques de responsabilisation. Ce langage est basé sur une logique formelle FOTL ce qui permet de vérifier la cohérence des politiques de responsabilisation ainsi que la compatibilité entre deux politiques. Les politiques sont ensuite traduites en une logique temporelle distribuée que nous avons nommée $FO\text{-}DTL^3$, cette dernière est associée à une technique de monitoring basée sur la réécriture de formules. Enfin nous avons développé un outil monitoring appelé AccMon qui fournit des moyens de surveiller les politiques de responsabilisation dans le contexte d'un système réel. Les politiques sont fondées sur la logique $FO\text{-}DTL^3$ et le framework peut agir en mode centralisée ou distribuée et fonctionne à la fois en ligne et hors ligne.

PUBLICATIONS

This might come in handy for PhD theses: some ideas and figures have appeared previously in the following publications:

[Ben15] Walid Benghabrit. « Framework pour la responsabilisation. » In: *Conférence en Ingénierie du Logiciel, CIEL 2015*. Bordeaux, France, June 2015.

[BGR16] Walid Benghabrit, Hervé Grall, and Jean-Claude Royer. *Monitoring accountability policies with AccMon framework*. GDR-GPL. Poster. June 2016.

[Ben+14a] Walid Benghabrit, Hervé Grall, Jean-Claude Royer, Mohamed Sellami, Monir Azraoui, Kaoutar Elkhiyaoui, Melek Önen, Anderson Santana De Oliveira, and Karin Bernsmed. « A Cloud Accountability Policy Representation Framework. » In: *CLOSER - 4th International Conference on Cloud Computing and Services Science*. Barcelone, Spain, Apr. 2014. URL: https://hal.inria.fr/hal-00941872.

[Ben+14b] Walid Benghabrit, Hervé Grall, Jean-Claude Royer, Mohamed Sellami, Karin Bernsmed, and Anderson Santana De Oliveira. « Abstract Accountability Language. » In: *IFIPTM - 8th IFIP WG 11.11 International Conference on Trust Management*. Singapore, Singapour, July 2014.

[Ben+14c] Walid Benghabrit, Hervé Grall, Jean-Claude Royer, and Mohamed Sellami. « Accountability for Abstract Component Design. » In: *EUROMICRO DSD/SEAA 2014*. Verona, Italy, Aug. 2014, pp. 213–220. DOI: 10.1109/SEAA.2014.68. URL: https://hal.inria.fr/hal-00987165.

[Ben+15a] Walid Benghabrit, Hervé Grall, Jean-Claude Royer, and Mohamed SELLAMI. « Abstract Accountability Language: Translation, Compliance and Application. » In: *ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE*. New Delhi, India, Dec. 2015. URL: https://hal.archives-ouvertes.fr/hal-01214365.

[Ben+15b] Walid Benghabrit, Hervé Grall, Jean-Claude Royer, and Mohamed Sellami. « Checking Accountability with a Prover. » In: *39th IEEE COMPSAC*. 2015, pp. 83–88. DOI: 10.1109/COMPSAC.2015.8. URL: http://dx.doi.org/10.1109/COMPSAC.2015.8.

[Ben+15c] Walid Benghabrit, Hervé Grall, Jean-Claude Royer, Mohamed SELLAMI, Monir Azraoui, Kaoutar Elkhiyaoui, Melek Onen, Santana Oliveira De Anderson, and Karin Bernsmed. « From Regulatory Obligations to Enforceable Accountability Policies in the Cloud. » In: *Cloud Computing and Services Sciences*. CLOSER 2014, Barcelona Spain, April 3-5, 2014, Revised Selected Papers. Springer International Publishing Switzerland, 2015. URL: https://hal.archives-ouvertes.fr/hal-01214387.

[SRB14] Mohamed SELLAMI, Jean-Claude Royer, and Walid Benghabrit. « Accountability for Data Protection. » In: *International Workshop on Computational Intelligence for Multimedia Understanding*. Paris, France, Nov. 2014. URL: https://hal.archives-ouvertes.fr/hal-01084890.

$$\Box(\forall x.\ x \in thanksList \Rightarrow thanks(x))$$

REMERCIEMENTS

"Nous sous-estimons trop souvent le pouvoir d'un contact, d'un sourire, d'un mot gentil, d'une oreille attentive, d'un compliment sincère, ou d'une moindre attention; ils ont tous le pouvoir de changer une vie."

— **Leo Buscaglia**

C'est avec grand plaisir que je remercie les personnes qui ont fait de ces années de thèse des années mémorables...

Albert Einstein a dit un jour que la valeur d'un homme tient dans sa capacité à donner et non dans sa capacité à recevoir. Durant ces années j'ai eu le grand honneur d'avoir travaillé avec un homme de valeur en la personne de mon directeur de thèse le Professeur Jean-Claude Royer. Un homme exemplaire et un chercheur dévoué et passionné par son métier. Merci à toi Jean-Claude pour m'avoir soutenu et pour m'avoir montré le chemin durant ces années de thèse.

Je tiens à remercier mon encadrant Hervé Grall pour tout le temps qu'il m'a consacré. Merci Hervé pour m'avoir soutenu, merci pour tes conseils et pour nos discussions passionnantes sur mon domaine de thèse, l'informatique, l'enseignement et la vie en général.

Un grand merci à Mario Südholt pour m'avoir accueilli dans son équipe ASCOLA, merci de m'avoir fait confiance pour enseigner le module MOE.

Je remercie les membres de l'équipe ASCOLA et mes collègues du département informatique pour m'avoir fourni un environnement de travail agréable et stimulant. En particulier Mohamed Sellami, Mehdi Haddad, Rémi Douence, Jaques Noyé, Adrien Lebre, Alexandre Garnier, Rémy Potier, Simon Dupont, Kevin Quirin, Simon Boulier et Hugo Brunelière. Merci à Philippe David qui m'a permis d'enseigner et de proposer des projets sur la méta programmation aux étudiants. Merci à Florence Rogues, Catherine Fourny, Soline Puente Rodriguez et Delphine Turlier, pour votre soutien administratif.

Un grand merci à mes collègues, co-bureau du B231 mais surtout amis Florant, Ronan et Johnatan, cette grande aventure a été très enrichissante et c'est en partie grâce à vous. Merci aux piliers d'arcadi Florent, Ronan et Jonathan ainsi qu'à tous les participants en particulier Amine, Alexandre, Gustavo, Ignacio, Ziad et Brice. Rien de

mieux pour décompresser que des ultra en "$\downarrow\rightarrow\downarrow\rightarrow PP\downarrow\rightarrow \ldots P\downarrow\rightarrow$" et des Shoryu dans le vide, et quand ça ne suffit pas, Ken Ω et Rachid. Je remercie l'équipe du club de robotique, Yann, benjamin, Thomas et Anqi pour le chemin que nous avons parcouru ensemble. Merci à Yann pour nos discussions techniques et philosophiques jusqu'à des heures pas possibles.

Je remercie également Roland Ducournau pour m'avoir encouragé à suivre la voie de la recherche, Clémentine Nebut et Pierre Pompidor pour m'avoir accueillis dans le master Aigle.

Je remercie Victor Gaugler et Ritta Baddoura pour la belle expérience d'enseignement qu'ils m'ont offert.

Merci à Ana Rosa Cavalli, Yves Roudier, Daniel Le Metayer, Nora Cuppens, Anderson Santana De Oliveira, pour m'avoir fait l'honneur d'être membre du jury de cette thèse.

Je remercie ma famille en particulier ma tante Aziza, ma grand-mère, mon oncle Chakib, et mes amis en particulier Aouni, Edi et Sylvie, pour m'avoir soutenu. Merci à Adel mon ami de longue date. Merci à Rémy, Véronique, Coralie, Arthur et Harold pour votre soutien et merci à Adélaïde pour m'avoir soutenue tout ce temps.

Enfin, je remercie ma mère, mon père et ma sœur Ismahane pour leur soutien, je vous en serai éternellement reconnaissant.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

AAL	Abstract Accountability Language
WEB	World Wide Web
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
SaaS	Software as a service
PaaS	Platform as a service
IaaS	Infrastructure as a service
FOTL	First Order Linear Temporal Logic
AccMon	Accountability Monitoring
AccLab	Accountability Laboratory
XACML	eXtensible Access Control Markup Language
PII	Personally identifiable information
CSP	Cloud service provider
GDPR	General Data Protection Regulation
GLBA	Gramm-Leach-Bliley Act
PHI	protected health information
PDRM	Personal Digital Rights Management
CNP	Colored Petri nets
APPEL	A-Posteriori PoLicy Enforcement
PLTL	Past Linear Temporal Logic
SIMPL	SIMple Privacy Language
OSL	Obligation Specification Language
IOT	Internet of Things
sloc	Source lines of code

Part I

ACCOUNTABILITY: BEYOND SECURITY AND PRIVACY

We live in a world where machines are omnipresent in our daily life, where the rooster was replaced by an alarm clock and smart phones, letters were replaced by emails and horses by cars with sophisticated electronics. Computers are everywhere, the 21st century human used them for leisure, working, communicating or even to control a spacecraft. In this world digital information is everywhere, and our personal data are exposed continuously, so what about our privacy? Our data are hosted in computers and travel via networks, is it sufficient to secure these components to protect our privacy? What about the human role in computer security and privacy protection? Does a perfect security guarantee our privacy?

"Perfect security is probably impossible in any useful system."

— **William D. Young**

1 INTRODUCTION

Contents

"Everybody with a computer should worry a little about whether hackers might break in and steal personal data."

— **John Viega**

1.1 Motivation

I discovered computer science by hacking when I was fourteen years old and I was surprised how easily we can get into a computer just by using hacking software and scripts without having a lot of knowledge in this field. Thinking of what a malicious script kiddie could do with these tools was scary.

In hacking culture, a script kiddie is an unskilled person who uses scripts or programs developed by other hackers to attack computer systems and networks.

I was convinced that an intensive control and a strong protection was the answer to all these security issues but at the same time I knew that every time security experts developed a new protection mechanism, hackers found new ways to bypass it. That's what made hacking funny for me but I was facing a paradox: As a hacker I couldn't assume that one day security experts would find the ultimate protection, but as person caring about my privacy I hope that they will find it and hackers could not break it.

My advisor told me that we would work on computer security and data privacy, I said to myself: We will find the ultimate protection! By enrolling in this thesis I was expecting to work on low-level security and develop new protection mechanisms and software, but it wasn't the case at all, at least at the beginning of my thesis. In the middle of my third year we started to work on low-level security tools and at that point I understood that if I had started directly on low level security something would have been missing. Indeed, when a person or a company violates our privacy and does not respect their privacy policy, we generally don't reply by hacking them, but we refer to law and judges. So the problem of privacy violation is not just a low level security issue, it also includes juridical, economical and social aspects.

But is it so important to protect our privacy in a digital world, after all, who cares?

Privacy concerns are related to information. The emergence of information technology gave a chance to peek at the private lives of strangers, thus people start to get concerned by their privacy. In the 21st century, the power of information is greater than ever, and privacy concerns are one of the biggest challenges for humanity today. Imagine a world without privacy, where our information and our privacy are shared everywhere with everyone. Will we be safe in this world? Privacy violations is a human problem, is it realistic to try to solve this issue using technical means? If we have a system in which violating privacy is technically impossible, is this system usable? And what if our privacy is controlled by an artificial intelligence? Should we consider a world without privacy?

I often had a conflicting discussion with my relatives and people in general concerning social networks, on-line Internet services that handle our personal data, computer security and privacy concerns. Below is a non-exhaustive list of some sentences that I heard:

— Privacy is not my concern, I have nothing to hide.
— Big companies know what they are doing, it is not in their benefit to violate the privacy of their users.
— But my photos are private and only my friends can see them.
— Who cares about me? I am not a famous person after all!
— I have the best anti-virus on the market and my computer is updated.
— No worries, I use private navigation mode.
— Yeah, but my data are processed by a machine, there is no human behind that.
— My anti-virus didn't find any virus.

So if you agree with one of these sentences, I hope that reading this manuscript will change your opinion!

1.2 Context and problematic

Accountability and the A4CLOUD project. Before I started my Ph.D, the first time that I saw the word *Accountability* in a computer science context was in the job description of this very thesis. My interviewer (who is now my supervisor) gave me a paper [EW07] of Sandro Etale to analyze.

In this paper there were two examples that illuminated my vision of accountability. Consider a school bus driver that takes the bus for his personal purpose, normally it is considered as a crime in New Orleans, but if his purpose was to help a neighborhood evacuate as Hurricane Katrina approached, it would probably have been recognized as heroic. Now if there had been a strict policy that prevents the bus driver to use the bus in other circumstances, he would not have been able to help the neighborhood. The second example is on criminal justice in open societies. Individuals are not controlled so tightly as to prevent

all crime. However, society is not overwhelmed by criminality because of the fear of legal consequences is a deterrent.

From these two examples we retain the following points:

— A strict security policy can sometimes hinder security.
— Relaxing security works only if we have responsibility and deterrence.

Accountability is defined in Black's Law Dictionary as follows: *"When one party must report its activities and take responsibility for them. It is done to keep them honest and responsible"*.

This thesis is in the context of the Cloud Accountability Project (A4Cloud). The goal of this project is to increase trust in cloud computing by devising methods and tools, through which cloud stakeholders can be made accountable for the privacy and confidentiality of information held in the cloud. These methods and tools combine risk analysis, policy enforcement, monitoring and compliance auditing. But how can we define policies that respect privacy? How can we check the compliance of different actors? And how to ensure that these policies are correctly implemented?

Talking to a computer. Humans use languages to communicate with each other, it generally works when the people speak the same language, but what about if two people speak different languages? In this case they need a translator to translate between the two languages. Human created machines and gave them the ability to talk binary language. In order to communicate with the machine, we can learn the binary language which is very fastidious to use, or the alternative is to make the machines understand our human language, which is quite difficult due to the number of spoken languages in the world and the ambiguity of human language. The intermediate solution is to create a language that is, easy to learn for a human, relatively simple to translate into machine language and that has more expressiveness power than binary language. We call such languages: *programming language*. Now having the binary language and our programming language we need a translator to translate the programming language into machine language, that what we call a *compiler*.

As a sentence in natural language can have many interpretations depending on the person who reads it, the same issue appears with programming languages. Thus, we need to agree on a common interpretation associated to a programming language that what we call *a semantic*.

1.3 Outline

This manuscript is divided into three main parts:

1. **Accountability: Beyond security and privacy**: this part is divided in two chapters:

- *Chapter 1 - Introduction*: the chapter that you are reading, this chapter introduces the motivation and the context of this thesis, and gives an overview of my vision regarding privacy and security questions.
- *Chapter 2 - State of art*: We will see why classical security mechanisms are not sufficient to protect our privacy, we will introduce privacy and accountability and we will present some logical background required for the contribution chapters.

2. **Accountability: from specification to implementation**: this part contains the contributions chapters and it is structured as follows:
   - *Chapter 3 - An abstract accountability language*: this chapter is about `AAL` which is a language designed to express accountability policy at a high level of abstraction. In this chapter we present the language and its semantic based on temporal logic, then we show how we can use `AAL` in order to check policies compliance and consistency. After that we present the principle of an accountable component diagram which is based on UML2 with `AAL` annotations, then we finish with some advanced features of `AAL` such as its type system and macros.
   - *Chapter 4 - Accountability enforcement*: in this chapter we will present a new logic called *FO-DTL*^{3} , a distributed first order temporal logic. Next we will present monitoring techniques for this logic, one using the progression technique and some optimizations and another one based on the inference rules called co-inductive monitoring. Finally, we will show how we monitor `AAL` policies using this *FO-DTL*^{3} monitor.
   - *Chapter 5 - Accountability tools*: in this chapter we will present the different tools developed in the context of this thesis. The first tool is an accountability framework called `AccLab`, the second one is a distributed monitoring engine that is used by `AccLab` for simulation and two other tools: (1) `AccMon` an accountability monitoring framework for web applications and (2) `PyMon` a python code monitoring framework.
3. **Accountability: Beyond accountability**
   - *Chapter 6 - Conclusion and future work*: in this chapter we will present the flow of reasoning that we had during this thesis and different key points concerning the future work.

2

STATE OF ART

Contents

"Think before you speak. Read before you think."

— **Fran Lebowitz**

2.1 Introduction

In this chapter we introduce the basic concepts and the background for this thesis. Since this thesis is in the context of the A4CLOUD project, first we present the cloud and the major concepts around it, then we talk about the Internet and the World Wide Web (WEB) which is the best known service offered by the Internet, we particularly focus on it and on security and privacy issues related to the Web. Then we talk about computer security and cyber-attacks and we discuss the limits of classical security mechanisms. Security does not guarantee the privacy, but without security there is no privacy at all. In the next section we talk about the privacy and how the new technologies impact the privacy in our daily life. Finally, we present the concept of accountability, its application in computer science and the different concepts required to follow the contribution part of this thesis.

2.2 The myth of computer security

2.2.1 The cloud, over the Internet and the Web

Cloud computing. In a October, 2009 a presentation entitled "Effectively and Securely Using the Cloud Computing Paradigm", by Peter Mell and Tim Grance of the National Institute of Standards and Technology (NIST) Information Technology Laboratory, cloud computing was defined as follows:

> *"Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable and reliable computing resources (e.g., networks, servers, storage, applications, services) that can be rapidly provisioned and released with minimal consumer management effort or service provider interaction."*

The key point of this definition is the externalization of resources. The literature defines three service models for the cloud depending on the type of shared resources:

For more details, the reader can refer to [MKL09]; [BBG11]

1. Software as a service (SaaS) delivers software applications over the Web.
2. Platform as a service (PaaS) delivers platforms to deploy custom applications over the web.
3. Infrastructure as a service (IaaS) rents processing, storage, network capacity, and other fundamental computing resources.

Sharing resources leads us to talk about their accessibility, there are three deployment models for the cloud:

1. Public Clouds generally provide resources over the Internet via web applications and web services, the cloud is hosted and managed by a third party.
2. Private Clouds act like private networks where resources are dedicated to a single organization.
3. Hybrid Clouds are composed with multiple public and private clouds.

2.2.1.1 The Internet and the World Wide Web

People often use the terms Internet and Web interchangeably.

The Internet is a system composed by interconnected computers that use the Internet protocols (TCP/IP). The WEB is probably the most well-known Internet service. The Web was initially designed as a distributed information system for scientists at CERN in 1990 and was proposed by Tim Berners-Lee and Robert Cailliau, at this time the web was an interconnected static hypertext documents relays on four components: `(1)` HyperText Markup Language (HTML) a standard markup language used to create web pages, `(2)` Hypertext Transfer

Protocol (HTTP) a protocol to transfer the web pages, (3) a web server which is a software used to serve the web pages, and (4) a web browser, a software to display the web pages. Nowadays the Web is a complex combination of many technologies and protocols ....

A recipe for disaster. From the point of view of a security researcher and a hacker I would define the Web as the following:

> *"A very very complex system, evolving at a very fast pace, designed to run arbitrary code and be reachable by everyone ... in the hands of amateurs who know nothing about programming..."*
>
> — **Davide Balzarotti**

Let us analyze this definition:

1. Very complex system: indeed, the web relies on the Internet which is composed of computers with different hardware running on multiple platforms with different operating systems. In addition there are many technologies used (communication protocols, browsers, plugins, etc).
2. Very fast evolution: many technologies appear and disappear during a short lapse of time, due to the market competition sometimes the security is just not taken in account in the earlier stage of development and sometimes these technologies are released without being matured.
3. Untrusted mobile code: that is what happens each time you visit a web page, your browser gets a piece of code developed by someone and runs it on your computer...
4. Attack surface: obvious since everything is interconnected means more entry points which leads to more potential attacks.
5. A platform for the masses: nowadays the democratization of the web makes it accessible to a large number of people of different categories.

Thanks to sandboxing techniques that helps a lot to minimize risks on this huge security whole ...

The last point is particularly interesting, indeed this facility to make a website by people with little experience in software development or with small/no awareness about security leads to have many vulnerable websites. But one could say why should we care about the security of Alice's blog that she uses to post her trips pictures ? Attackers do not care about her website and even if her blog is hacked, it's just a simple blog ... To answer this question we have to see the types of attacks on the Web and the goal of attackers. We distinguish two types of attacks: the targeted and non-targeted attacks. Non-targeted attacks are mostly automated and widespread, attackers generally use a tool that perform automated scans and once the tool identify a vulnerable target, another tool tries to exploit the vulnerability and installs a backdoor. On the other hand targeted attacks are mostly manual and

target specific sites or companies, attackers spent a lot of time studying and analyzing their target and they do not use automated tools to remain undetectable, once everything is ready they perform the attack. The targeted attacks can be easily justified, the goal of attackers can be social or political activism, cyberwars, personal vengeance, or to gain money using Web ransom-ware [1] But to understand the goal of non-targeted attacks we have to investigate what attackers do with these websites after attacking them. Studies like [Ala+06] using honeypots showed that targets attacked by non-targeted attacks are mostly used for phishing, scams and to perform targeted attacks. So the issue is when a security breach in Alice's blog allowed to perform an attack on a governmental organization, who is responsible? The attacker, the framework used by Alice to develop her blog, the web server provider, or Alice?

Global architecture of the Web. The figure 1 shows the global view of the Web architecture in a general manner. The client interacts with a computer (1), and sends a request to a server using a web browser, the browser runs under an operating system (2), the request is transmitted to the internal router (5) and then to a proxy which routes the request (11). The reverse proxy routes the request to the destination server (10), then the request is handled by the web application (6) which interacts with an operating system (7), a database (8) and a backup server (9). Finally, a response is sent to the client from the web application (6) to his web browser (3) which potentially runs some plug-ins (4).

Sadly each node of the web architecture presented in figure 1 represents a potential attack vector. There are several techniques to bypass security mechanisms such as network sniffing, identity spoofing, keyloggers, etc. But the most effective attacks are social engineering based ones since the user remains the weakest point of the chain.

2.2.2 Cyber attacks and Cyber defense

In order to understand the importance and the stakes of security in the 21st century we need to study the impact of cyber security on our real world.

The economic problem. Many years ago, hackers looked for vulnerabilities in systems for popularity and for fun, then they disclose the vulnerabilities publicly. Disclosure was a way to counter poorly-written software at the time where security concerns and awareness were not priority for developers, since the objectives of the companies were to produce software quickly and sell their products. But today the world

1. https://www.htbridge.com/blog/ransomweb_emerging_website_threat.html

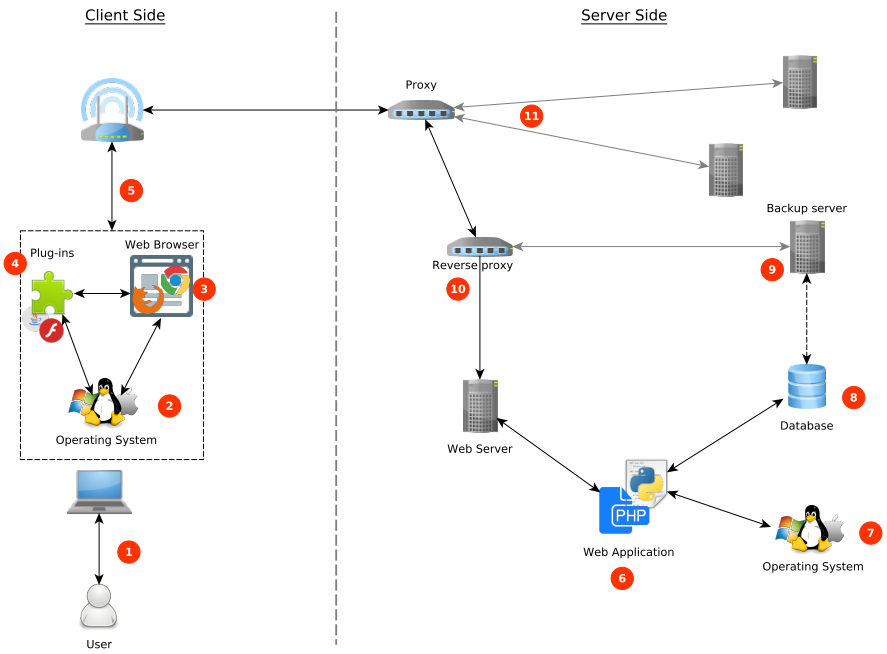


Figure 1 – Global Web architecture

has changed, the disclosure of vulnerabilities that we hear about are just the visible part of the iceberg. The hidden part is now on the black/gray market which becomes a huge business. Companies offer rewards to who finds security threats in their systems in order not to disclose the vulnerabilities and the worst is that in many cases they do not fix the vulnerabilities immediately due to economic issues. Other companies are specialized in selling zero-day vulnerabilities (a security hole that is unknown to the owner) to cyber criminals.

The social problem. We should not ignore the importance of the human part in computer security, as we say, the bug is between the chair and the keyboard. Real and targeted attacks relies mostly on social engineering, a pirate spends most of the time to prepare the attack by collecting information about the target, his contacts, his habits, etc. Even in academics we teach more how to write programs than how to write a secure code. Security should be a concern of every person who writes a piece of code.

For more details on social engineering, we recommend [MS03]; [MS06]

The technical problem. Computer science is an exact science in theory, but in real life there are many factors that make things unpredictable. Bugs are often a source of vulnerabilities, and bugs that are introduced by humans may remain hidden for many years before being discovered even if the software source code is public (like Heartbleed [Hea] which is a famous vulnerability in the open source world).

2.3 Privacy, an illusion in a digital world

For all people who told me that they do not care about privacy because they have nothing to hide, I would like to answer them by this quote:

"Arguing that you don't care about the right to privacy because you have nothing to hide is no different than saying you don't care about free speech because you have nothing to say."

— **Edward Snowden**

In my opinion, not caring about privacy because we think that we have nothing to hide is similar to thinking that we cannot be hacked because we have installed the best anti-virus on the market... A naive and meaningless way of thinking which harms the society. I don't know which is the worst, don't care about privacy or think that we have the control of privacy? What about public cameras, mobile phones, public networks, drones, etc? Nowadays we can forget our privacy on the Internet, as John Viega said:

"By now , people should have a reasonable expectation that there's no privacy on the Internet."

— **John Viega**

2.3.1 But what is privacy about?

The concept of privacy varies widely among countries, cultures, and jurisdictions. It is shaped by public expectations and legal interpretations; as such, a concise definition is elusive if not impossible. Privacy rights or obligations are related to the collection, use, disclosure, storage, and destruction of personal data (or Personally identifiable information (PII)). Personal data means any information relating to an identified or identifiable natural person.

As mentioned in [MKL09], privacy advocates have raised many concerns about cloud computing. These concerns typically mix security and privacy. Here are some additional considerations to be aware of.

- **Access:** Data subjects have a right to know what personal information is held and, in some cases, can make a request to stop processing it. In the cloud, the main concern is the organization's ability to provide the individual with access to all personal information, and to comply with stated requests. If a data subject exercises this right to ask the organization to delete his data, will it be possible to ensure that all of his information has been deleted in the cloud?
- **Compliance:** What are the privacy compliance requirements in the cloud? What are the applicable laws, regulations, standards,

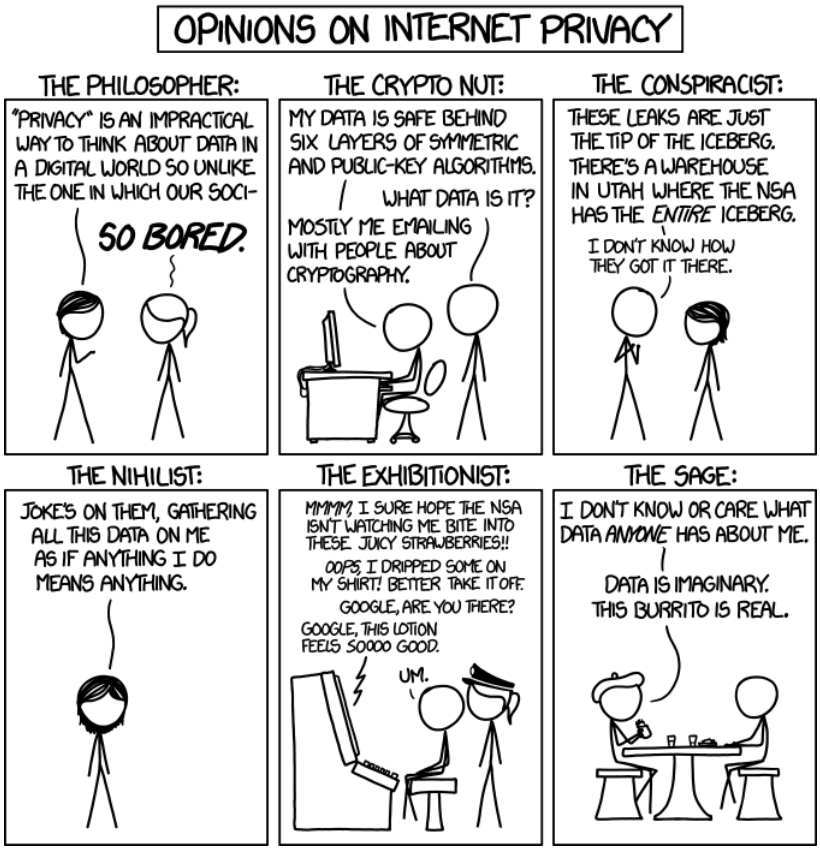


Figure 2 – Privacy opinions - XKCD

and contractual commitments that govern this information, and who is responsible for maintaining the compliance?

— **Storage:** Where is the data in the cloud stored? Was it transferred to another data center in another country? Is it commingled with information from other organizations that use the same cloud service provider? Privacy laws in various countries place limitations on the ability of organizations to transfer some types of personal information to other countries. When the data is stored in the cloud, such a transfer may occur without the knowledge of the organization, resulting in a potential violation of the local law.

— **Retention:** How long is personal information retained? Which retention policy governs the data? Does the organization own the data, or the cloud service provider? Who enforces the retention policy in the cloud?

— **Destruction:** How does the cloud provider destroy PII at the end of the retention period? How do organizations ensure that their PII is destroyed by the Cloud service provider (CSP) at the right point and is not available to other cloud users? How do they know that the CSP didn't retain additional copies? Did the

CSP really destroy the data, or just make it inaccessible to the organization?

— **Audit and monitoring:** How can organizations monitor their cloud service provider and provide assurance to relevant stakeholders that privacy requirements are met when their PII is in the cloud?

— **Privacy breaches:** How do you know that a breach has occurred? How do you ensure that the CSP notifies you when a breach occurs? Who is responsible for managing the breach notification process (and costs associated with the process)?

2.3.2 Privacy in the real world: laws and regulations

Legal aspects for data privacy vary across the world, some countries apply strict enforcement of privacy directives whereas other countries have no privacy directives. This makes it challenging for multinational companies that process personal data of customers from different jurisdictions. When processing personal data in a global environment, a major challenge is the fact that some requirements are conflicting. The jurisdiction of privacy laws and directives is determined differently in different countries and states. Some of the laws are based on the location of the organization, some on the physical location of the data centers, and some on the location of the data subjects. The only universal consistency is that the law has not caught up with the technology [MKL09].

There is an international trend in protecting data, for instance the Gramm-Leach-Bliley Act (GLBA) [US] tackles the privacy in the financial world. The Financial Privacy Rule requires financial institutions to provide their customers with a privacy notice upon inception of the relationship and annually. The privacy notice must explain information collection, sharing, use, and protection. The HIPAA rules [Sch+08] regulates the use and disclosure of protected health information (PHI) by health care providers and health plans. It stipulates the right of an individual to access his PHI and have any inaccuracies corrected. The HIPAA Privacy Rule requires health care providers to notify individuals of their information practices. However, with data being in the cloud, those notices may be incomplete or inaccurate.

In Europe, privacy is considered a basic human right and cannot be divorced from one's personal freedom. The EU 95 Directive [Dir95] (replaced now by General Data Protection Regulation (GDPR)) [EU] oblige member states to implement and enforce data privacy legislation that satisfies the requirements present in the EU Directive. The EU Directive distinguishes between data controllers and data processors. It is the controller that is responsible for implementing an effective mechanism and, therefore, ensuring its use of compliant third-party service providers.

The GDPR is applied to the processing of personal data, whether it is automated (even partially) or not. The GDPR defines principles regarding data and processing [Lau17]. Data principles consists in:

— **Transparency:** data must be processed fairly, lawfully and transparently.
— **Purposes:** data should only be collected for determined, explicit and legitimate purposes, and should not be processed later for other purposes.
— **Minimization:** the data processed must be relevant, adequate and limited to what is necessary in view of the purposes for which they are processed.
— **Accuracy:** the data processed must be accurate and up-to-date regularly.
— **Retention:** the data must be deleted after a limited period.
— **Subject explicit consent:** the data may be collected and processed only if the data subject gives his explicit consent.

2.3.3 Towards a formalization of privacy

There are several work that tackle privacy issues using formal methods. In [MÓ9] the authors present a restricted natural language SIMple Privacy Language (SIMPL) to express privacy requirements and commitments. Their work is part of a broader multidisciplinary project which follows a top-down approach, starting from the legal analysis and defining technical and legal requirements for the development of an effective solution to privacy issues. Another example is the privacy language of [BMB10] provides a formal and decidable language to write data privacy policies and close to natural language.

Other works handle abstract privacy policy like in [BA05] the authors describe a general process for developing semantic models from privacy policy goals mined from policy documents. In [KL03], the authors develop an approach where contracts are represented by XML documents enriched with logic metadata and assistance with a theorem prover. DeYoung et al in [DeY+10] define Privacy Least Fixed Point: a formal language to express data privacy management. They demonstrate the ability of this language by representing the obligations of two consequent sets of rules. The readability for non-expert user is, of course, a problem, since it is based on fixed points (this is similar to the language of mCRL2).

The data Privacy Logic [PD11] is a deontic and linear temporal logic based on predicates dealing with personal data management. The authors focus on obligations with deadlines and maintained interdiction, that is on deontic logic with time. They identifed requirements for such a privacy language and surveyed most of the existing candidates and found them insufficient. Thus they introduce two new operators: one

for obligation with deadline and another one for maintained interdiction and they illustrate their use.

In [MGL06], authors present an approach that consists in extending the access control matrix model to deal with privacy rules which were used to express the HIPAA [Hea96] consent rules in a formal way and to check properties of different versions of the HIPAA. However, this model does not deal with sticky policies, compliance or future obligations as in [MÓ9].

In [GMS05] authors used the same access control matrix approach to express privacy policies for location-based services based on a Personal Digital Rights Management (PDRM) architecture.

The Obligation Specification Language (OSL) [Hil+07] is illustrated through DRM policies but can also be used to express privacy policies. It includes usage requirements such as duration, number of times, purpose, notification, etc. It also includes different modalities (such as *must* and *may*) and temporal operators.

2.4 Accountability: beyond security to preserve privacy

Previous work on security showed that it is not sufficient to ensure data privacy in distributed systems where new agents may be involved and can send data to others. Several articles already mention the limits of the pure security approach (for instance [Wei+08]) and suggest relying on the concept of accountability. Indeed, we live in an interconnected world where information can be easily copied, aggregated and automated correlations across multiple databases uncover information even when it is not revealed explicitly.

Thus, there is a recent interest and active research for accountability which overlaps several domains like security [Wei+08]; [ZX12]; [PW13], language representation [MÓ9]; [DeY+10], auditing systems [Fei+12]; [Jag+09], evidence collection [SSL12]; [Hae+10] and so on. However, only few of them consider an interdisciplinary view of accountability taking into account legal and business aspects.

Regarding tool supports and frameworks we can find several proposals [Wei+09]; [Hae+10]; [ZWL10]. But none of them provides a holistic approach for accountability in the cloud, from end-user understandable sentences to concrete machine-readable representations. In [SSL12], authors propose an end-to-end decentralized accountability framework to keep track of the usage of the data in the cloud. They suggest an object-centered approach that packs the logging mechanism together with users' data and policies.

Recently Butin et al. advocate for strong accountability in [BCM14]. The authors consider that the view from normative text put too much emphasis on accountability of policy and procedure while computer scientists have a too narrow approach of accountability by practice. They put forward strong accountability as a set of precise legal obligations

supported by an effective software tool set. They demonstrate that the state of the art in terms of technology is sufficient to ensure the notion of accountability by design. The authors while they promote a precise and operational approach think that a multidisciplinary view of accountability is fundamental. This interdisciplinary approach is necessary since a pure computer science approach is not sufficient to tackle some contractual aspects. Laws and normative texts contain many features we cannot represent with computers or we do not want that computer decide without human control.

There are theoretical formal models like [Ced+05]; [EW07]; [Jag+09] but they do not provide concrete languages and means to verify accountability at design time.

The A-Posteriori PoLicy Enforcement (APPEL) core presented in [EW07] combines an audit logic with trust management techniques. It makes it possible to define sticky data policies; in addition, it includes provisions for constraining the join of documents and defining policy refinement rules. Trust and accountability are central and the formal setting is based on audit logs.

2.4.1 Policy languages: towards a link between worlds

Existing data privacy languages such as [DeY+10]; [PD11]; [BMB10]; [Cho+13] only consider access or usage control, they did not address full accountability. However these approaches provide advanced comparisons for permissions and obligations.

Several approaches to contract specification define specific modalities generally classified in permission or authorization, obligation and prohibition. There are mainly three approaches: relying on deontic logic as in [PD11], using a pure temporal approach, or a mix of both, for instance [BBF06]. Deontic logic appears as a natural choice. However it is subject to few confusing paradoxes like the Good-Samaritan, the Knower, the Contrary-to-duty paradox and the SecondBest Plan [Cas81]; [HPT07]. Note that there are several work that try to solve these paradoxes [Fel90], In [Cho+13] the authors present a policy language based on a restricted subset of FOTL. The restricted subset of FOTL allows some past operators, but it limits future operators in specific places and the decision procedure is based on a small finite model theorem. Another related work is [TDW13] which focuses on purpose restrictions governing information use. This paper provides a formal semantics for purpose restrictions which is based on simulating ignorance of the information prohibited. Policy languages can be classified into four categories:

1. *Access Control:* eXtensible Access Control Markup Language (XACML, [OAS13]);

2. *Privacy:* The Platform for Privacy Preferences (P3P, [P3p]), the Primelife Policy Language (PPL, [Ard+09]) and SecPal for Privacy (SecPal4P, [BMB10]);
3. *Policy specification for security:* Conspec ([AN08]) and Ponder ([Dam+01]);
4. *Service Description:* The Unified Service Description Language (USDL, [BO12]), SLAng ([LSE03]) and WS-Policy ([OAS06]; [OAS12]).

2.4.2 Temporal logic: formalizing a physical world

Propositional logic. Propositional logic is a simple logic that aims to study propositions which can be `true` or `false`. These propositions can be composed with each other using connectors (called logical operators) such as the disjunction (`or`), conjunction (`and`), conjunction (`and`), material condition (`if...then`) and the negation (`not`). Bellow Table 1 shows the syntax of the propositional logic.

ψ	::=	`true` | `false`	(`constant symbol`)
	|	$\bigcirc\psi \mid \neg\psi \mid \psi(\vee \mid \wedge \mid \Rightarrow \mid \Leftrightarrow)\psi$	(`propositional formula`)
	|	P	(`propositional symbol`)

Table 1 – Propositional Logic syntax

The interpretation for a well-formed formula is to assign either a truth value $\top$ or a falsehood value $\bot$.

I	$\vDash$	`P`	=	`literal`
I	$\vDash$	$\neg\psi$	=	`not` ψ
I	$\vDash$	$\psi_1 \wedge \psi_2$	=	$I \vDash \psi_1$ `and` $I \vDash \psi_2$
I	$\vDash$	$\psi_1 \vee \psi_2$	=	$I \vDash \psi_1$ `or` $I \vDash \psi_2$
I	$\vDash$	$\psi_1 \Rightarrow \psi_2$	=	`if` $I \vDash \psi_1$ `then` $I \vDash \psi_2$
I	$\vDash$	$\psi_1 \Leftrightarrow \psi_2$	=	$I \vDash (\psi_1 \Rightarrow \psi_2)$ `and` $I \vDash (\psi_2 \Rightarrow \psi_1)$

Table 2 – Propositional logic semantics

Satisfiability and validity. A formula ψ is satisfiable if there is an interpretation I that satisfies ψ. In the same way, if there is no interpretation that satisfies ψ the formula is unsatisfiable. For example, the formula $p \wedge q$ is satisfiable whereas $p \wedge false$ is unsatisfiable. A formula is valid if all interpretations I satisfies ψ. For example the formula $p \vee true$ is valid.

For more details on temporal logic and on formal methods in general we recommend the excellent book by Fisher [Fis11].

Temporal logic. Temporal logic is an extension of classical logic that introduces operators relating to time. Temporal logic often contains operators such as $\bigcirc$, meaning in the next moment in time, $\Box$, meaning at every future moment, and $\Diamond$, meaning at some future moment, $\psi_1 \mathcal{U} \psi_2$, meaning that ψ_1 has to hold at least until ψ_2 holds, $\psi_1 \mathcal{R} \psi_2$, meaning that ψ_2 has to hold until ψ_1 holds (including the point where ψ_1 holds).

ψ	::=	$\texttt{true} \mid \texttt{false} \mid \neg\psi \mid \psi(\vee \mid \wedge \mid \Rightarrow)\psi \mid \varphi$	(`propositional formulas`)
	$\mid$	$\bigcirc\psi \mid \Box\psi \mid \Diamond\psi \mid \psi\mathcal{U}\psi \mid \psi\mathcal{R}\psi$	(`temporal formulas`)
φ	::=	$P(t^*)$	(`predicates`)

Table 3 – Temporal Logic syntax

Model-checking and proofs. As defined in [Fis11], the proof theory of a logic consists of axioms and inference rules (alternatively called proof rules). Axioms describe 'universal truths' for the logic and inference rules transform true statements into other true statements (theorems). The process involved in a proof is generally that of transforming such true statements in the logic using only the inference rules and axioms. Technically, a proof is a sequence of formulae, each of which is either an axiom or follows from earlier formulae by a rule of inference.

On the other hand, model-checking consists in checking if there are execution sequences of a given system that do not satisfy the required formula, then at least one 'failing' sequence will be returned as a counter-example. If no such counter-examples are produced then this means that all executions of the system satisfy the prescribed formula.

Part II

ACCOUNTABILITY: FROM SPECIFICATION TO IMPLEMENTATION

In this part we present our work which is about filling the gap between the specification (how things should work) and implementation (how things really work). In the first chapter we present a methodology to take in account accountability at design time, we will show how to make an accountable design and present accountable design principles. Then we present a language called *AAL* (Abstract Accountability Language) that allows to specify Accountability policies and a specification methodology around this language.

In the second chapter of this part we will talk about accountability monitoring, in order to enforce the specifications we synthesize a reference monitor for each actor in the system, the role of the monitors is to ensure that the specified policies are respected.

In the last chapter we present the different tools developed during this thesis . We present a tool *AccLab* (Accountability Laboratory) that provide assistance for writing and checking policies and also provide a simulation platform to see accountability in action. Next we talk on monitoring tools such as *AccMon* (Accountability Monitoring) that allows to monitor accountability policies in the context of a real system. This tool has an interconnection with *AccLab*.

"A good policy language allows not only the expression of policy but also the analysis of a system to determine if it conforms to that policy. The latter may require that the policy language be compiled into an enforcement program (to enforce the stated policy) or into a verification program (to verify that the stated policy is enforced)."

— **Matt Bishop**

3

AN ABSTRACT ACCOUNTABILITY LANGUAGE

Contents

"The development of policy languages focuses on supplying mathematical rigor that is intelligible to humans."

— **Matt Bishop**

3.1 Introduction

As said before, there is a big gap between laws and regulations which are written in natural language and the real implementation of systems. In addition writing policies without a tool that checks their consistency can lead to unexpected behaviors. Thus in this chapter we present a formal model for accountability in order to tackle these issues. This model relies on a language called AAL which is a language that allows to write accountability policies. First we introduce the `AAL` language with its syntax and we illustrate its usage with some examples. After that we present the FOTL that is the semantics of the `AAL` language, and the interpretation of `AAL` over `FOTL`. Next we talk about the compliance, the consistency and how we verify these properties on `AAL` policies based on the `FOTL` translation. Then we present an approach to handle accountability at design time using extended UML component diagram. Before we conclude, we talk on the advanced usage of `AAL` with the

macros and type system. Finally, we conclude this chapter and we discuss the benefits and the limitations of our approach. Note that all examples used in this chapter are available in the examples folder of AccLab [1].

3.2 Abstract Accountability Language (AAL)

According to [Fei+12] accountability can be fulfilled in five temporal steps: prevention, detection, evidence collection, judgment and punishment. In order to represent accountability obligations, we adopt the same point of view by synthesizing these steps in three parts : (1)*usage control* for the preventive aspects, (2)*audit* for detection, evidence collection and judgment, and (3)*rectification* for punishment/compensation.

3.2.1 AAL language syntax

We consider an `AAL` clause as a triplet *(ue, ae, re)* which informally means: *In any execution state, do the best to ensure the usage expression (ue), and if a violation of the usage is observed by an audit (ae) then the rectification (re) applies.* A clause is identified by a name and composed with at least a usage expression in which we define the policy that we want to ensure,

an audit part in which we specify actions related to the audit phase (such as the auditor, the audit frequency, etc) and a rectification part in which we define what we should do if a violation occurs on the usage part. Bellow the template for an `AAL` clause:

```
CLAUSE policy_name (
    <usage_expression>
    AUDITING <audit_expression>
    IF_VIOLATED_THEN <rectification_expression>
)
```

The core of expressions are actions, the syntax of an action is the following:

```
// action
actor1.service1'['actor2']'(parameters) [PURPOSE purpose]
```

Which means that 'actor1' uses a service called 'service1' which is provided by 'actor2' using the arguments 'parameters'. This vision is inspired by the architectural design of a system in a component diagram [RJB04]. The `PURPOSE` keyword is optional and is used to denote that the action is used for the specific purpose 'purpose'.

In addition to these actions we introduce the permission and prohibition which is common in security and access control languages [OAS13]; [EW07]. The `AAL` syntax is as follows:

1. https://github.com/hkff/AccLab See Chapter 5 for more information.

```
PERMIT action  // Permission
DENY   action  // Prohibition
```

Actions and authorizations (permissions/prohibitions) can be composed with different boolean operators (`AND`, `OR`, `NOT`) and conditions (`IF(...) THEN {...}`).

Example 1. The following example illustrates the basic usage of `AAL`. We want to express the following policy : *"Alice can read her data that are managed by Bob, John cannot read Alice's data, and if a violation occurs when Bob is audited Bob should inform Alice about the violation."*

```
CLAUSE alice_policy (
    // Allow alice to read 'aliceData' from bob
    PERMIT alice.read[bob](aliceData) AND
    // Deny john to read 'aliceData' from bob
    DENY john.read[bob](aliceData)
    // The agent auditor audit the agent bob
    AUDITING auditor.audit[bob]()
    // If a violation occurs, bob should notify alice about the violation
    IF_VIOLATED_THEN bob.notify[alice]("Data access violation")
)
```

One important point when we express policies and behaviors in general is the notion of time, we need means to express that something will always or never occur or will occur in the future, and also precedence between events (i.e. eventA occurs before eventB), thus we introduce the following temporal operators: (`ALWAYS`, `NEVER`, `SOMETIME`, `UNTIL`, `UNLESS`). The first three operators are unary and are used as follows: `Operator expression.` `ALWAYS` and `NEVER` mean that expression will always/never happens, `SOMETIME` means that the expression will occur at some point in the future. `UNTIL` and `UNLESS` are binary operators and they are used as follows: `expression_{1} Operator expression_{2}.` `UNTIL` means that $expression_1$ doesn't occur before $expression_2$, and `UNLESS` means that while $expression_2$ does not occur, $expression_1$ have to.

In addition to linear discrete time (expressed in `AAL` using previously presented operators), `AAL` proposes a limited way to express real-time constraints using predicates expressing dates. Real-time operates only on atomic actions and not on composed expressions.

```
// Real time action
Action [BEFORE | AFTER time]
```

Here `time` is a string that represents time in days/months/years (for example "2 days", "3 months", "1 year").

We also need a means to express quantifier constructions that are present in natural language (e.g. Aristotle syllogism), thus introduce the quantification operators (`FORALL` and `EXISTS`).

$$\forall x.(Human_x \Rightarrow Mortal_x \wedge Human_{Socrates}) \Rightarrow Mortal_{Socrates}$$

```
FORALL x:Type  // Universal quantifier
EXISTS x:Type  // Existential quantifier
```

Example 2. Let's enrich the previous example with the new concepts:

```
CLAUSE alice_policy (
      // For all data where the subject is alice
      FORALL x:Data
      IF(x.subject == alice) THEN {
              // Allow alice to read her data from bob
          PERMIT alice.read[bob](x) AND
          // Deny john to read alice's data from bob
          DENY john.read[bob](x) AND
          IF(alice.requestDeletion[bob](x)) THEN {
              SOMETIME(bob.delete[bob](x))
          }
      }
      AUDITING auditor.audit[bob]()
      // If a violation occurs, bob should notify alice about the
   violation
      IF_VIOLATED_THEN bob.notify[alice]("Data access violation")
)
```

Note that some elements are optional, details are presented in Listing 1.

We need to declare the different actors of the system with their services and the used data. Bellow the AAL syntax of declarations.

```
AGENT   name TYPES(type1 ...) REQUIRED(service1 ...)
             PROVIDED(service1 ...)
DATA    name TYPES(type1 ...) REQUIRED(service1 ...)
             PROVIDED(service1 ...)
SERVICE name TYPES(type1 ...)
```

Example 3. To complete the previous example with declarations:

```
AGENT alice TYPES(User) REQUIRED(read) PROVIDED(notify)
AGENT bob TYPES(Provider) REQUIRED(read notify) PROVIDED()
AGENT john TYPES(User) REQUIRED(read) PROVIDED()
SERVICE read
DATA aliceData TYPES(Data)
```

Finally, we want to express properties on actors and objects, for that we introduce a special operator (@) called predicate.

```
      // Predicate syntax in AAL
      @<predicate_name>(parameters)
```

Example 4. In this example we illustrate the use of predicates. Aristotle syllogism: All men are mortal and Socrates is a man, therefore Socrates is mortal.

```
AGENT Socrates
CLAUSE syllogism (
      ALWAYS(
        FORALL x:Human @Mortal(x) AND
        IF (@Human(Socrates)) THEN { @Mortal(Socrates) }
      )
)
```

With these constructions we cover all data privacy concepts such as access control, data retention, data transfer, etc.

We presented the main concepts of AAL language which allow you to write basic AAL programs. The minimal grammar of AAL in listing 1

regroups all the concepts presented previously. The grammar is written here in a pseudo Backus–Naur form. `::=` represents a definition of an expression, `|` a disjunction, `*` a repetition, `[...]` an optional expression, `{ }` are used for grouping, and `//` for comments.

Listing 1 – AAL core grammar

```
// A. AAL program
AALprogram  ::= {Declaration | Clause}*

// B. Declaration part
Declaration ::= AgentDec | ServiceDec | DataDec
ServiceDec  ::= SERVICE Id TYPES(type*)
AgentDec    ::= AGENT Id [TYPES(type*)]
                         [REQUIRED(service*) PROVIDED(service*)]
DataDec     ::= DATA  Id TYPES(type*)
                         [REQUIRED(service*) PROVIDED(service*)]

// C. Clause part
Clause         ::= CLAUSE Id(Usage [Audit Rectification])
Usage          ::= ActionExp
Audit          ::= AUDITING ActionExp
Rectification ::= IF_VIOLATED_THEN ActionExp

// D. Action expression (usage, audit and rectification expressions)
ActionExp      ::= Action | NOT ActionExp | Modality(ActionExp)
                 | Condition | ActionExp1 BinaryOp ActionExp2
                 | Author | Quant ActionExp | Predicate
                 | IF (ActionExp1) THEN '{' ActionExp2 '}'

// E. Action expression components
Action         ::= agent1.service'['agent2']'(Exp) [Time] [Purpose]
Time           ::= {AFTER | BEFORE} date
Purpose        ::= PURPOSE(Id)
Exp            ::= Variable | constant | Id.attribute | Predicate
Predicate      ::= @Id(arg*)
Author         ::= {PERMIT | DENY} Action
Condition      ::= [NOT] Exp | Exp1 {== | !=} Exp2
                 | Condition1 {AND | OR} Condition2
Quant          ::= {FORALL | EXISTS} Variable
Variable       ::= Id : type
Modality       ::= ALWAYS | NEVER | SOMETIME | NEXT
BinaryOp       ::= AND | OR | UNTIL | UNLESS
Id, type, service, agent, arg, constant, attribute, date ::= literal
```

an AAL program is composed of a set of clauses and declarations.

Next we present the semantics of our language and we show how an AAL program is interpreted.

3.2.2 Temporal logic: semantics for AAL language

In order to interpret our language AAL, we use the linear temporal logic [Fis08] with quantified data. FOTL is not a decidable logic, nor a semi-decidable [Fis11] but there are several fragments with decidable properties. We particularly focus on the **monodic** fragment [HWZ00] which supports a semi-decidable decision method. Such method is pre-

free variable : a variable that is used by a quantifier

sented in [LH10] and has been implemented in a theorem prover TSPASS that we use in our implementation. The monodic condition states that any temporal formula has, at most, one free variable.

e.g. this formula is not monodic : $\forall x.\exists y.G(P_x \wedge P_y)$

whereas those formulas are monodic : $\forall x.\exists y.G(P_x)$ $G(\forall x.\exists y.P_x \wedge P_y)$

The syntax of FOTL is described in Table 4. We recall that for readability the operators : $\bigcirc$, $\Box$, $\Diamond$ (respectively `next, always, future`) are often written `X, G` and `F`.

ψ	::=	`true` | `false` | $\neg\psi$ | $\psi(\vee \mid \wedge \mid \Rightarrow)\psi$ | φ	(`propositional formulas`)
	|	$\exists x.\psi$ | $\forall x.\psi$	(`first-order formulas`)
	|	$\bigcirc\psi$ | $\Box\psi$ | $\Diamond\psi$ | $\psi\mathcal{U}\psi$ | $\psi\mathcal{R}\psi$	(`temporal formulas`)
φ	::=	$P(t^*)$	(`predicates`)

Table 4 – First Order Linear Temporal Logic syntax

Compared to LTL, FOTL introduces the universal quantifier ($\forall$) and the existential quantifier ($\exists$). For detailed semantics of LTL/FOTL the reader can refer to [Fis11]; [Dix+07]; [Pnu77].

Interpreting AAL expressions over FOTL. AAL language is close to FOTL which makes the translation straightforward. Indeed, boolean/temporal operators, predicates, and constants are directly translated into FOTL.

Definition 1. Let I the interpretation function of AAL expressions over FOTL which is inductively defined as follows :

An action is interpreted as a predicate: $service(actor1, actor2, I(\Delta \vDash exp))$ where the parameter can be either a reference to quantified variable, a constant or a predicate which has a direct translation into FOTL. An action with a purpose is interpreted using the implication between the interpretation of the action and the purpose (the action implies the purpose). A typed variable is interpreted as a predicate $type(id)$.

For expressions with a quantified variables the translation depends on the quantifier operator. With a universal quantifier, the expression is interpreted using an implication: $\forall var.id.I(\Delta \vDash var) \Rightarrow I(\Delta \vDash Exp)$. For example, the following AAL expression `FORALL x:Data alice.read[bob](x)` is translated as follows: $\forall x.Data(x) \Rightarrow read(alice, bob, x)$. For the existential quantifier we use the duality with the universal quantifier which gives: $\exists var.id.I(\Delta \vDash var) \wedge I(\Delta \vDash Exp)$.

The expression `IF exp1 THEN exp2` is translated with a simple implication: $I(\Delta \vDash exp1) \Rightarrow I(\Delta \vDash exp2)$. Finally for each action we add the corresponding authorization link that states that if we have an action we have the permission for this action:

```
(always (![x, y, z] (action(x, y, z) => Paction(x, y, z)) ))
```

$I(\Delta$	$\vDash$ `literal`$)$	$=$	`literal`
$I(\Delta$	$\vDash$ `agent1.service[agent2](Exp)`$)$	$=$	`service(agent1, agent2,`$I(\Delta \vDash$ `Exp`$))$
$I(\Delta$	$\vDash$ `NOT ActionExp`$)$	$=$	$\neg I(\Delta \vDash$ `ActionExp`$)$
$I(\Delta$	$\vDash$ `Modality(ActionExp)`$)$	$=$	$I(\Delta \vDash$ `Modality`$)(I(\Delta \vDash$ `ActionExp`$))$
$I(\Delta$	$\vDash$ `ActionExp1 BinaryOp ActionExp2`$)$	$=$	$I(\Delta \vDash$ `ActionExp1`$)I(\Delta \vDash$ `BinaryOp`$)I(\Delta \vDash$ `ActionExp2`$)$
$I(\Delta$	$\vDash$ `PERMIT Action`$)$	$=$	$\mathtt{P}I(\Delta \vDash$ `Action`$)$
$I(\Delta$	$\vDash$ `DENY Action`$)$	$=$	$\neg\mathtt{P}I(\Delta \vDash$ `Action`$)$
$I(\Delta$	$\vDash$ `Action PURPOSE(p)`$)$	$=$	$I(\Delta \vDash$ `Action`$) \Rightarrow \mathtt{P}$
$I(\Delta$	$\vDash$ `FORALL Variable. ActionExp`$)$	$=$	$\forall I(\Delta \vDash$ `Variable`$) \Rightarrow I(\Delta \vDash$ `ActionExp`$)$
$I(\Delta$	$\vDash$ `EXISTS Variable. ActionExp`$)$	$=$	$\exists I(\Delta \vDash$ `Variable`$) \wedge I(\Delta \vDash$ `ActionExp`$)$
$I(\Delta$	$\vDash$ `Id(arg1 .. argN)`$)$	$=$	`Id(arg1,..argN)`
$I(\Delta$	$\vDash$ `IF(ActionExp1) THEN {ActionExp2}`$)$	$=$	$I(\Delta \vDash$ `ActionExp1`$) \Rightarrow I(\Delta \vDash$ `ActionExp2`$)$
$I(\Delta$	$\vDash$ `Id:Type`$)$	$=$	`Id(Type)`
$I(\Delta$	$\vDash$ `Id.attribute`$)$	$=$	`attribute(Id)`
$I(\Delta$	$\vDash$ `Condition1 AND Condition2`$)$	$=$	$I(\Delta \vDash$ `Condition1`$) \wedge I(\Delta \vDash$ `Condition2`$)$
$I(\Delta$	$\vDash$ `Condition1 OR Condition2`$)$	$=$	$I(\Delta \vDash$ `Condition1`$) \vee I(\Delta \vDash$ `Condition2`$)$
$I(\Delta$	$\vDash$ `NOT Exp`$)$	$=$	$\neg I(\Delta \vDash$ `Exp`$)$
$I(\Delta$	$\vDash$ `Exp1 == Exp2`$)$	$=$	$\mathtt{EQUAL}(I(\Delta \vDash$ `Exp1`$), I(\Delta \vDash$ `Exp2`$))$
$I(\Delta$	$\vDash$ `Exp1 != Exp2`$)$	$=$	$\neg\mathtt{EQUAL}(I(\Delta \vDash$ `Exp1`$), I(\Delta \vDash$ `Exp2`$))$
$I(\Delta$	$\vDash$ `Modality`$)$	$=$	`Modality`
$I(\Delta$	$\vDash$ `BinaryOp`$)$	$=$	`BinaryOp`
$I(\Delta$	$\vDash$ `AgentDec`$)$	$=$	$\mathtt{always}(\wedge^{i}_{1..n}$ `Ti(Id)`$_{T \in \mathtt{AgentDec.TYPES}})$
$I(\Delta$	$\vDash$ `DataDec`$)$	$=$	$\mathtt{always}(\wedge^{i}_{1..n}$ `Ti(Id)`$_{T \in \mathtt{DataDec.TYPES}})$
$I(\Delta$	$\vDash$ `ServiceDec`$)$	$=$	$\mathtt{always}(\forall$`x,y,z.Id(x, y, z)`$\Rightarrow$ `PId(x, y, z))`

Table 5 – AAL semantics over FOTL

In addition, translating `AAL` expressions needs to generate some context. First we generate an invariant for the equality predicate in order to use the comparison between actors:

```
// Identity
(always ![a] (Actor(a) => EQUAL(a, a))) &
// Comparison
(always ![a, b]
((Actor(a) & Actor(b) & EQUAL(a, b)) => EQUAL(b, a)))
```

Next we generate knowledge for the types:

```
// Type habitation
(?[a] Type(a)) & (?[a] always(Type(a)))
// Subtyping relation
(?[a] Type(a) & (![x] SubType(x) => Type(x)))
```

Finally, for each declared actor we add subject:

```
(?[d] subject(d, actorName))
```

Note that our equality is defined without transitivity property due to FOTL theoretical limitations.

The interpretation function I always terminates if the program is syntactically correct since all expressions lead to atoms.

Example 5. Bellow the translation into FOTL of the AAL program presented in example 3.

```
%%% Action authorizations
always (
( ![x, y, z] (notify(x, y, z) => Pnotify(x, y, z)) ) &
( ![x, y, z] (read(x, y, z) => Pread(x, y, z)) )
) &


%%% Actors knowledge
always (
// AGENT alice TYPES(User) REQUIRED(read) PROVIDED(notify)
( User(alice) ) &
// AGENT bob TYPES(Provider) REQUIRED(read notify) PROVIDED()
( Provider(bob) ) &
// AGENT john TYPES(User) REQUIRED(read) PROVIDED()
( User(john) )
)
&

// FORALL x:Data
((![x] ( Data(x) =>
  // IF(x.subject == alice) THEN
  ( ((subject(x, alice)) =>
      // PERMIT alice.read[bob](x) AND
      (((Pread(alice, bob, x) &
      // DENY john.read[bob](x) AND
            ~Pread(john, bob, x)) &
      // IF (john.read[bob](x)) THEN
      ((read(john, bob, x)) =>
            // SOMETIME( bob.notify[alice]("Data access") )
            (sometime(notify(bob, alice, CTS0))))))))) )) ))
```

AAL clause semantics. In this part we provide the interpretation of an AAL clause *(ue, ae, re)*.

In a given state there are two cases: either the usage is satisfied or violated and in this last case if an audit happens a rectification should be done. This leads to the following formula:

$$I(\Delta \vDash \texttt{Clause(ue, ae, re)}) =$$
$$G(I(\Delta \vDash ue) \vee (\neg I(\Delta \vDash ue) \wedge (G(I(\Delta \vDash ae) \Rightarrow I(\Delta \vDash re)))))$$

This is not the unique possible interpretation, many variations on the rectifications or the audit events are possible. For instance, the rectification can hold few times after violation occurs.

$I(\Delta \vDash \texttt{Clause(ue, ae, re)}) =$
$G(I(\Delta \vDash ue) \vee (\neg I(\Delta \vDash ue) \wedge (G(I(\Delta \vDash ae) \Rightarrow F(I(\Delta \vDash re))))))$

3.2.3 Expressing security and privacy in AAL

AAL addresses privacy and security concepts described in Chapter 2. Authorizations are expressed using permissions in AAL, and obligations using SOMETIME action in AAL. The purpose is explicitly managed using PURPOSE keyword in actions.

Concerning data retention, data transfer, user consent and notifications, these concepts can be handled using AAL actions, since AAL provides actions based on uninterpreted predicates.

For example, data retention is handled using the delete action with a time period. The following example means that *"the Hospital should delete Kim's data within two months"*.

Listing 2 – Data retention example

```
1 ALWAYS (FORALL a:Data
2    IF(d.subject==Kim) THEN {
3               hospital.delete[hospital](d) BEFORE "2 months"})
```

Data transfer and location controls are easily expressed as soon as we defined geographic areas as a partition of types. Once declared a type in AAL the prefix @ denotes the associated type predicate. The example below states that *"Hospital is allowed to transfer only to European countries"*.

Listing 3 – Data transfer example

```
1 ALWAYS (FORALL d:Data FORALL target:Agent
2    IF(@Europe(target)) THEN {PERMIT Hospital.transfer[target](d)})
3 AND
4 ALWAYS (FORALL d:Data FORALL target:Agent
5    IF(NOT @Europe(target)) THEN {DENY Hospital.transfer[target](d)})
```

Listing 4 – Data subject explicit consent example

```
1 FORALL d:data
2    IF(d.subject==Kim) THEN {
3      (NOT Hospital.process(d)) UNTIL Kim.giveConsent[Hospital](d)}
```

Listing 5 – Breaches notification example

```
1 CLAUSE kim_policy (
2    FORALL d:data
3       IF (d.subject==Kim) THEN {
4              PERMIT Kim.read[bob](aliceData) AND ... }
5    AUDITING auditor.audit[hospital]()
6    IF_VIOLATED_THEN Hospital.notify[Kim]("Data access violation")
7 )
```

3.3 Handling accountability at design time

Historically in software development security issues were addressed at the end of the software development life cycle, probably by lack of time or means, this leads to situations where security issues are not patchable at all. Thus it is important to handle security issues at an earlier stage of the development life cycle, and the same remark is valid for privacy and accountability. Note that we make a distinction between accountability at design time and accountability by design. An accountable system by design requires extra specially on the logs collection for the audit [BLM14]. In this part we consider the design of accountability systems. We propose to define a system's design based on components diagram.

3.3.1 Accountable design

We enrich classic component design notations from UML V2, and we propose a set of guidelines to evaluate the suitability of the design regarding accountability. A system is composed of components representing agents or processes. These components provide services (called *provided services*) and require services (called *required services*) from other components. We attach an `AAL` clause to each component.

Principle for accountable design. The main principle for our accountable component design which is also shared with rules in security and privacy policies, is that a clause associated to a component/service must respect the data's user preference. The rationale for this is that, in any execution state, the clause must logically imply the user preferences. To check the compliance of two accountability contracts we rely on the following sufficient condition : if the provider provides stronger usage, audit and rectification expressions than those expected by the required side then the contract compliance is ensured. Furthermore, as data can travel along a communication path (through several services in a component design), we need a strict evolution. It means that along communication paths, the accountability clauses are strengthened, otherwise ensuring the user preferences will fail. Note that we do not consider merge and split of data, we assume that data can be tracked from its owner to any other agent which gets it. But we allow multiple data and multiple paths in the component design.

We first state the main principle, thus the notion of accountable component design as following:

Definition 2. (Preferences principle). The user preferences of any data should be ensured by all the services of the component design.

The strict evolution of accountability clauses can be defined as follows.

Definition 3. (Well-Connected property). In a well-connected component design, on any communication a provided clause always is compliant with the connected required clause. Also, in case where communication cycles exist in a component design, well-connection implies that all clauses associated to the services in the cycle are logically equivalent.

Definition 4. (Preferences property). In a well-connected component design, if the user preferences are compliant with the provided clause at the input of a component design then the preferences principle is satisfied. The preferences property provides guarantees on the data circulating in the component design.

3.4 Checking and proving accountability

3.4.1 Model-checking versus proofs

In order to check to properties of an accountable design, our first approach was to use the model-checking technique.

We rely on the `mCRL2` toolset [Cra+13] and a translation of the component design as well as the clauses. We choose this approach (rather than the PrivacyLFP [DeY+10] or the Data Privacy Logic [PD11]) because there is an efficient toolset to support various analyses. The `mCRL2` language is devoted to the formal specification and analysis of distributed systems [Gro+08]. The language is a process algebra allowing data type specification in a functional style. The language supports the expression of complex temporal properties involving time and data. The `mCRL2` branching temporal logic is based on the least and greatest fixed points and allows quantifications on data. A toolset [Cra+13] for specification, state space exploration and verification by model-checking is available. It also provides the use of parameterized Boolean equation system. From a specification and a formula expressing a property, the toolset builds a parameterized Boolean equation system which can be solved using the prover of the toolset.

We successfully implemented this approach, for more details the reader can refer to [Ben+14]. However using a model-checker will force us to define a more operational behavior for the agents. In addition, we do not need a specific computation model and we must check for the validity of clauses, that means clauses will be satisfied in any implementation model. Thus, our second approach is the use of the theorem proving technique. We rely on the `TSpass` tool which is a prover supporting the `FOTL` monodic class. It provides a semi-decision procedure for satisfaction and validity. Regarding efficiency, the theoretical resolution complexity behind `TSpass` is elementary [HWZ00]. Despite this, practical experiments have been successfully achieved with `TeMP` in [FG+06] and `TSpass` is competitive and outperform `TeMP` on unsatisfiable problems [LH10]. Furthermore, [SD11] compares it with several classic LTL model-checkers and sat solvers and concludes that no one

dominates or solves all instances. This approach is presented in the next section.

3.4.2 Clauses consistency and compliance

A component diagram can be translated into `AAL`, the components are translated as actors with their corresponding provided and required services. Policies attached to actors The links between services/components are translated into validity checks.

After writing a clause we want to check if the clause is consistent, and does not contain for instance a permission and a prohibition on the same action. Then we want to check various properties on the clause for example if an action or a set of action is permitted by the clause. Finally, we want to compare two or more clauses, and check if our clause is compliant with another.

Definition 5 (Clause consistency)**.** Let C be a context and P an AAL expression, proving that P is consistent is equivalent to check if $C \wedge P$ is satisfiable.

Definition 6 (Clauses compliance)**.** Let C be a context, P_1 and P_2 an AAL expressions, proving that P_1 is compliant with P_2 is equivalent to check that $C \Rightarrow (P_1 \Rightarrow P_2)$ is valid. In order to check the validity we apply the following checks:

1. $C \wedge (P_1 \wedge P_2)$ is satisfiable.
2. $C \Rightarrow (P_1 \Rightarrow P_2)$ is satisfiable.
3. $\neg(C \Rightarrow (P_1 \Rightarrow P_2))$ is unsatisfiable.

Example 6. Here is an example for the satisfiability and compliance of three policies. The first one `bob_policy` is not satisfiable due to the `PERMIT` and `DENY` on the same action with the same parameters (line 4). The second and the third clauses `provider_policy` and `alice_policy` are both satisfiable but not compliant with each other. The problem here is that in the `provider_policy` we deny every actor from reading his data (line 11) and in `alice_policy` we allow alice to read her data (line 20).

```
1  CLAUSE bob_policy (
2       FORALL x:Data
3       PERMIT alice.read[bob](x) AND DENY alice.read[bob](x)
4  )
5  CALL sat_ue("bob_policy") // bob_policy clause is not satisfiable
6
7  CLAUSE provider_policy (
8       FORALL x:Data FORALL y:Actor
9       IF(x.subject == y) THEN {
10        DENY y.read[bob](x) AND
11        DENY john.read[bob](x)
12      }
13 )
```

Note the use of a new keyword CALL to run macros which is explained in 3.5.

```
14 CALL sat_ue("provider_policy") // provider_policy is satisfiable
15
16 CLAUSE alice_policy (
17      FORALL x:Data
18      IF(x.subject == alice) THEN {
19        PERMIT alice.read[bob](x) AND
20        DENY john.read[bob](x) AND
21        IF (john.read[bob](x)) THEN {
22          SOMETIME( bob.notify[alice]("Data access") )
23        }
24      }
25 )
26 CALL sat_ue("alice_policy")// alice_policy is satisfiable
27 // provider_policy is not compliant with alice_policy
28 CALL validate_usage("provider_policy" "alice_policy")
```

3.4.3 Conflict detection and localization

One of the limits of this approach is the difference between the natural language and the logic language which is very tricky. We know that the natural language is not logic and is subject to multiple interpretations. For instance, one source of misinterpretation is the logical implication ($\Rightarrow$) which is the translation of `IF THEN` `AAL` operator. Let's demonstrate it with a concrete example.

Example 7.

```
1 CLAUSE bob_policy (
2      FORALL x:Data
3      IF(@login(alice)) THEN {
4        PERMIT alice.read[bob](x) AND DENY alice.read[bob](x)
5      }
6 )
7 CALL sat_ue("bob_policy") // bob_policy clause is satisfiable
```

Here we expect the consistency check on `bob_policy` clause to be unsatisfiable, since we can easily see that we have a permission and a prohibition on the same action, but the check will return a satisfiable result! This is due to the semantics of the implication, if `@login(alice)` false there is no need to check the expression inside the `THEN`. In order to detect such conflicts, we have some heuristics that apply different checks using variations on the context. For example, the case of the logical implication, we run a check by forcing the conditions to be true.

This is derived from the truth table of the logical implication, $False \Rightarrow ?$ is always True.

Another point about conflict detection is that we want to localize and isolate the expression that makes the consistency or compliance checks fail. We use two techniques: the first one is the masking technique as in [Sch12], in which we mask sub-formulas consecutively in order to isolate the conflict. The second one is the combining technique, in which we combine a sub-formula with the rest of the clause.

Let us consider the following `AAL` example:

Example 8.

```
/*** Kim's user preference ***/
CLAUSE kim_policy (
  FORALL file:data
  IF(file.subject == kim) THEN {
   PERMIT kim.read[cloudX](file)
   AND cloudX.delete[file]() BEFORE "2 Years"
   AND IF (cloudX.delete[file]()) THEN {
    cloudX.notify[kim]("DATA DELETED")}
 }
 AUDITING DPA.audit[cloudX]()
 IF_VIOLATED_THEN MUST(DPA.sanction[cloudX]())
)
/*** CloudX's policy ***/
CLAUSE cloudX_policy (
  FORALL a:agent FORALL file:data
  IF(file.subject == a) THEN {
   DENY a.read[cloudX](file)
   AND cloudX.delete[file]() BEFORE "3 Years"
  }
  AUDITING DPA.audit[cloudX]()
  IF_VIOLATED_THEN DPA.sanction[cloudX]()
)
```

Using the logical prover shows that the clause cloudX policy is not consistent with Kim's clause, but no more information is provided. Using our tool `AccLab` that implements combining and masking technique precisely show to the user the following conflicts:

- `PERMIT kim.read[cloudX](file)` doesn't match with `DENY a.read[cloudX](file)`.
- `cloudX.delete[file]() BEFORE "2 Years"` doesn't match with `cloudX.delete[file]() BEFORE "3 Years"`.
- `IF (cloudX.delete[file]()) THEN cloudX.notify[kim]("DATA DELETED")` is not guaranteed.

3.5 Advanced usage of AAL

We introduced some extensions to **AAL** for usability and practical issues. Before we explain in detail the extensions, we present below the complete grammar of **AAL** in Listing 6.

Listing 6 – AAL Grammar

```
// A. AAL program
AALprogram ::= Declaration | Clause | Comment
             | Macro | MacroCall | Loadlib | LtlCheck
             | CheckApply | Exec | Env | Behavior

// B. Declaration part
Declaration ::= AgentDec | ServiceDec | DataDec | TypesDec | VarDec

ServiceDec  ::= SERVICE Id [TYPES(Type*)] [Purpose]
AgentDec    ::= AGENT Id [TYPES(Type *)
```

```
                          [REQUIRED(Service*) PROVIDED(Service*)]
DataDec     ::= DATA Id TYPES(Type*)
                          [REQUIRED(Service*) PROVIDED(Service*)]
VarDec      ::= Type Id [attribute(value*)]*

// C. Clause part
Clause        ::= CLAUSE Id(Usage [Audit Rectification])
Usage         ::= ActionExp
Audit         ::= AUDITING ActionExp
Rectification ::= IF_VIOLATED_THEN ActionExp

// D. Action expression (usage, audit and rectification expressions)
ActionExp     ::= Action | NOT ActionExp | Modality (ActionExp)
                | Condition | ActionExp1 BinaryOp ActionExp2
                | Author | Quant* ActionExp | Predicate
                | IF (ActionExp1) THEN '{' ActionExp2 '}'

// E. Action expression components
Action        ::= agent1.service['['agent2']'](Exp) [Time] [Purpose]
Purpose       ::= PURPOSE(Id*)
Exp           ::= Variable | constant | Id.attribute | Predicate
Predicate     ::= @Id(parg*)
Author        ::= {PERMIT | DENY} {Action | ActionExp}
Condition     ::= [NOT] Exp | Exp1 {== | !=} Exp2
                | Condition1 {AND | OR} Condition2
Quant         ::= {FORALL | EXISTS} Variable [WHERE Condition]
Variable      ::= Id : Type
Modality      ::= ALWAYS | NEVER | SOMETIME | NEXT | MUST | MUSTNOT
BinaryOp      ::= AND | OR | ONLYWHEN | UNTIL | UNLESS
Time          ::= {AFTER | BEFORE} date

// F. Type system extension
TypesDec      ::= TYPE Id [{EXTENDS | UNION | INTERSECT}(Type*)]
                        [ATTRIBUTES(Id*)] [ACTIONS(Id*)]

// G. Reflexion extension
Macro         ::= MACRO Id [(param*)] (MCode)
MCode         ::= '"""' Metamodel api + target interpreter
                        language code '"""'
MCall         ::= CALL Id(arg*)
LoadLib       ::= LOAD STRING
Exec          ::= EXEC MCode
                                                        Macro execution.
// H. FOTL checking extension
LtlCheck      ::= CHECK Id(Check)
Check         ::= '"""'FOTL formula + clause(Id)[.ue | .ae | .re]'"""'
CheckApply    ::= APPLY Id()
                                                        Direct call to LTL
                                                        prover.
// I. Utils
Env           ::= ENV MCode
// J. Templates
Template      ::= TEMPLATE Id (param*) (ActionExp)
Behavior      ::= BEHAVIOR Id (ActionExp)

Type, var, val, attr Id, agent, Constant, date ::= literal
```

3.5.1 Usability and development extensions

AAL is a research language thus was designed some extensions to make it developer-friendly in order to experiment new ideas easily.

Libraries. For readability and modularity, we may need to split our AAL program into several separate files, thus we introduced the ability to load external AAL files into another one. To load a library we use the following syntax: `LOAD "dir1.dir2....dirN.aal_file"`. We exploit this mechanism to define a standard library for AAL language, which provides many useful constructions, such as macros and predefined types (see below).

Note that '/' are replaced by '.' and the file extension is omitted.

Macros. Macros allow to manipulate the AAL language and to perform different kinds of operations and checks on an AAL program. There are several macros defined in AAL standard library that comes with AccLab[2]. Here is an example of a macro `clause_fotl` which is present in the standard library and how we call it in an AAL program:

Note that the internal macro language used here is the language of the AAL interpreter in AccLab which is written in Python3.4.

```
/*
 * clause_fotl
 * :param c: a string that represents a clause name
 * :effect: shows the FOTL translation
 */
MACRO clause_fotl(c) (
"""
cl = self.clause(c)     // Get a clause by name
if cl is not None:      // If the clause exists
    print(cl.to_fotl())  // Show the clause FOTL translation
"""
)

// Call the macro on alice_policy
CALL clause_fotl("alice_policy")
```

Here is a more interesting example, we want to check that all actions are called with a purpose.

```
MACRO check_purpose ()(
 // Getting all used actions in the program
 actions = self.aalprog.walk(filter_type=m_aexpAction)
 // We loop over the actions
 for x in actions:
   if x.purpose is None:
     // If there is no purpose in the current action, we print a
    warning message
     print("Warning : Action " + str(x.action) + " is used without a
    purpose !")
)
```

Another possibility is to execute a code directly while the program is being parsed. The syntax is as follows:

MACROS and EXEC are analogous to functions and lambda (without parameters).

2. See Chapter 5 for more details.

```
EXEC """ Same as the code used inside macros """
```

Checks. In an AAL program we can directly include FOTL formulas which can be checked by the prover. In addition to FOTL, the following commands are available:

- clause(c): will be replaced by the FOTL translation of the clause c.
- clause(c).[ue/ae/re]: will be replaced by the FOTL translation of the usage/audit/rectification part of the clause c.
- @verbose: shows the final translated formula.

Bellow a compliance check example:

The syntax used for FOTL is the syntax accepted by the prover.

```
// LTL check declaration
CHECK chk1 (
  clause("kim_policy").ue => clause("cloudX_policy").ue
)

// Call the check
APPLY chk1()
```

Context. As the same principle of checks, the keyword ENV allows to add FOTL formula into the translation context which is used when the prover runs compliance and consistency checks.

Templates. In order to improve AAL usability for non specialists in formal methods, we go further in assisting privacy and security officers in writing policies with the help of dedicated templates.

A template, or a behavioral pattern, is a function which transforms several AAL expressions into a more complex AAL sentence expressing a specific accountability practice. In other words, it allows to replace simple predicates with complex AAL expressions.

The keyword TEMPLATE allows to define an expression, the predicate @template allows to refer to a given template and the predicate @arg allows to refer to a template argument.

```
// Template
TEMPLATE acc (ue:Behavior, ae:Behavior, re:Behavior) (
  ALWAYS( @arg(AE) AND ALWAYS (@arg(UE) OR
    ((NOT( @arg(UE))) AND (ALWAYS ( IF(@arg(AE)) THEN { @arg(RE)}))))))
)

// Expressions
BEHAVIOR b1 (
    FORALL d:data FORALL a:Actor
    // Allow users to read their data
    IF (d.subject == a) THEN {
        PERMIT a.read[css](d) AND DENY a.read[css](d)
    } AND
)
```

```
BEHAVIOR b2 (
  dpa.audit[css](log)
)

BEHAVIOR b3 (
  MUST(dpa.sanction[css](d) AND css.delete[css](d))
)

// Template instanciation
@template(acc, b1, b2, b3)
```

The template and the expressions can be stored in a separate AAL files, and the instantiation will be done using the load keyword `LOAD "template file"`.

3.5.2 Extending authorizations

There are many cases where we need to specify complex authorizations which do not cover only actions but a chain of actions. For instance we can allow some actions but at the same time prohibit a specific chaining of these actions.

We extend our grammar as following:

```
Author        ::= {PERMIT | DENY} {Action | ActionExp}
```

For instance consider the following example :

Example 9. We want to allow alice to modify secret data, to send secret data but not to send modified secret data.

```
FORALL d:Secret
PERMIT alice.modify(d) AND
PERMIT alice.send[bob](d) AND
DENY (FORALL d:Secret alice.modify(d) AND SOMETIME alice.send[bob](d))
```

We extend our semantics to have two kinds of authorization, a sub world for authorizations that are applied to expressions; and another one for atomic actions.

Definition 7. The interpretation function I of AAL expressions over FOTL is extended as follows:

For each action we add the corresponding authorization link that states that if we have an action we have the permission for this action:

```
(always (![x, y, z] (action(x, y, z) => Paction(x, y, z)) ))
```

For each complex permission `PERMIT (ActionExp)` we generate a context rule that links complex permissions and atomic permissions. This rule has the following form:

```
// link between complex and atomic permissions
always (GPermit(ActionExp) => LPermit(ActionExp))
```

The function GPermit copies the ActionExp by replacing each action with a unique permission (GPAction) in the global authorizations world.

The function `LPermit` copies the `ActionExp` by replacing each action with a unique permission (PAction) in the atomic authorizations world.

Next, we generate a context rule that links between expressions and permissions:

```
// link between expressions and permissions
always (ActionExp => GPermit(ActionExp))
```

Finally, the complex permission is translated as follows:

```
GPermit(ActionExp)
```

$$
\begin{array}{rcl}
I(\Delta \models \texttt{PERMIT ActionExp}) & = & GPermit(\texttt{ActionExp}) \\
I(\Delta \models \texttt{DENY ActionExp}) & = & \neg I(\Delta \models \texttt{PERMIT ActionExp}) \\
GPermit(\Delta \models \texttt{Action}) & = & \texttt{GP}I(\Delta \models \texttt{Action}) \\
GPermit(\Delta \models *) & = & I(\Delta \models *)
\end{array}
$$

Table 6 – AAL extended authorizations semantics over FOTL

The translation of the example in Listing 9 gives:

```
(always
  (![x, y, z] (modify(x, y, z) => Pmodify(x, y, z))) &
  (![x, y, z] (send(x, y, z) => Psend(x, y, z)))
) &
 // link between complex and atomic permissions
(always
  ![d] (Secret(d) => GPmodify(alice, alice, d) & sometime(GPsend(alice,
     bob,d)))
  =>
  ![d] (Secret(d) => Pmodify(alice, alice, d) & sometime(Psend(alice,
    bob,d)))
) &
// link between expressions and permissions
(always
  ![d] (Secret(d) => modify(alice, alice, d) & sometime(send(alice, bob
    ,d)))
  =>
  ![d] (Secret(d) => GPmodify(alice, alice, d) & sometime(GPsend(alice,
     bob,d)))
) &
// FORALL d:Secret
(![d] (Secret(d) =>
    // PERMIT alice.modify(d) AND PERMIT alice.send[bob](d) AND
    Pmodify(alice, alice, x) & Psend(alice, bob, d) &

    // DENY (FORALL d:Secret alice.modify(d) AND SOMETIME alice.send[
    bob](d))
    ~(![d] (Secret(d) => GPmodify(alice, alice, d) & sometime(GPsend(
    alice, bob,d))))
  )
)
```

3.5.3 AAL type system

As mentioned before, agents, services and data should be typed in AAL with literals which actually refer to declared types in the program. A type is defined by a unique identifier, a set of super types, a set of attributes and a set of actions.

```
TYPE id EXTENDS(supertype1...supertypeN)
      ATTRIBUTES(attribute1...attributeN) ACTIONS(action1...actionN)
```

The typing relation EXTENDS is translated in FOTL using the implication.

```
// Type habitation
?[a] (type(a)) &
// Subtyping relation
![x] (type(x) => supertype1(x) & type(x)...supertypeN(x))
```

Note that we can also define the negation, conjunction, and union of type only with the EXTENDS operator using the implicational propositional calculus. We need only to define the bottom type (false) and the top type (true). For more readability we introduced two operators UNION and INTERSECT.

```
TYPE True
TYPE False

// T = Not A
TYPE A EXTENDS(False)
TYPE T EXTENDS(A)

// T = A & B
TYPE T INTERSECT(A B)

// T = A | B
TYPE T UNION(A B)
```

Bellow the semantics of the UNION and INTERSECT operators:

$$
\begin{aligned}
I(\Delta \models \texttt{TYPE T INTERSECT(A B)}) &= (A \Rightarrow (B \Rightarrow False)) \Rightarrow False \\
I(\Delta \models \texttt{TYPE T UNION(A B)}) &= (A \Rightarrow B) \Rightarrow B
\end{aligned}
$$

Table 7 – AAL union and conjunction type semantics over FOTL

Type checker is too strict, disabling it by default is a design choice.

In order to use a type checker, you have to add the following macro call: CALL enable_type_checker() to your program. Before we present the type system, let's take a look at the following example:

```
TYPE Data
TYPE Secret EXTENDS(Data)
TYPE Public EXTENDS(Data)
SERVICE send TYPES(Secret)
AGENT alice
AGENT bob
```

```
CLAUSE c1 (
      // Allow alice to send to bob any public data
      FORALL d:Public PERMIT alice.send[bob](d)
)
```

This AAL program is syntactically correct but semantically wrong, in the sense that here we allow to use the send service with public data but the service declaration shows that we can only use it with secret data. Such kinds of errors can be detected easily and statically using a type checker and without translating AAL into FOTL with its overhead.

Thus we introduce a simple type system to AAL and we focus mainly on actions Action and expressions Exp.

```
T  ::= A     // Agent type
     | S     // Service type
     | E     // Expression
```

$$\text{Action}\ \frac{\Gamma \vdash \texttt{s} : \texttt{S} \quad (s \subseteq_R \texttt{agent1}) \wedge (s \subseteq_P \texttt{agent2}) \qquad \Gamma \vdash \texttt{e} : \texttt{E} \quad e \subseteq_S \texttt{s}}{\Gamma \vdash \texttt{Action} : \texttt{S + E}}$$

$$s \subseteq_R a\ \frac{\Gamma \vdash \texttt{s} : \texttt{S} \qquad \Gamma \vdash \texttt{a} : \texttt{A} \qquad (\texttt{s} \in \texttt{a.required}) \vee (\texttt{s} \in \texttt{a.types.actions})}{\Gamma \vdash s \subseteq_R \texttt{a} : \texttt{A + S}}$$

$$s \subseteq_P a\ \frac{\Gamma \vdash \texttt{s} : \texttt{S} \qquad \Gamma \vdash \texttt{a} : \texttt{A} \qquad (\texttt{s} \in \texttt{a.provided}) \vee (\texttt{s} \in \texttt{a.types.actions})}{\Gamma \vdash s \subseteq_P \texttt{a} : \texttt{A + S}}$$

$$e \subseteq_S s\ \frac{\Gamma \vdash \texttt{e} : \texttt{E} \qquad \Gamma \vdash \texttt{s} : \texttt{S} \qquad \exists\ \texttt{t/t} \in \texttt{e.types} \wedge \texttt{t} \in \texttt{s.types}}{\Gamma \vdash e \subseteq_S \texttt{s} : \texttt{E + S}}$$

$$\text{Variable}\ \frac{\Gamma \vdash \texttt{Type} : \texttt{T}}{\Gamma \vdash \texttt{Variable} : \texttt{T}} \qquad \text{Id.attribute}\ \frac{\Gamma \vdash \texttt{Id} : \texttt{T} \qquad \texttt{attribute} \in \texttt{T.attributes}}{\Gamma \vdash \texttt{Id.attribute} : \texttt{T}}$$

Table 8 – AAL type system

An action which has the form agent1.service[agent2](Exp) is well typed if: (1) the service is declared in the required services of agent1 or is an action in the types of agent1; (2) the service is declared in the provided services of agent2 or is an action in the types of agent2; (3) the type of Exp is declared in types of service. An Id.attribute expression is well typed if attribute is declared in the attributes of Id types.

The advantages of the type system are that we can detect and localize precisely typing errors which may not be localized by our conflict localization techniques. In addition we avoid the computational overhead introduced by the translation to FOTL and prover runs.

When we apply typing rules to the previous example, the type checker reports the following errors:

1. Agent alice uses the service send which is not required

2. Agent alice uses the service send which is not provided by agent bob
3. The service send is called with a non-compatible argument d. Expected: { Secret } Found: { Public|Data }.

In order to solve type errors, first we need to complete the declarations:

```
AGENT alice TYPES(Data) REQUIRED(send) PROVIDED()
AGENT bob TYPES() REQUIRED() PROVIDED(send)
```

For the third error, either we make the service `send` supports the type `Public` `SERVICE send TYPES(Secret Public)` or we change the type of the quantified variable to `Secret` or its subtypes `FORALL d:Secret`.

Typing the full `AAL` program is a part of our future work. This may contribute to refining the conflict localization.

3.6 Conclusion and discussion

We presented AAL which is a formal language, close to natural language that allows to express accountability policies. Several languages exist for specifying privacy preferences and policies, but some of them are less readable for non-specialist users (e.g. lawyers, privacy officers), some of them lack a formal modal and others doesn't cover all accountability requirements.

For instance the language eXtensible Access Control Markup Language (XACML) based on XML language and `AAL` are quite generic on resources and subjects. XACML allows also the use of generic actions (AnyAction), which can be easily implemented in `AAL` using macros. Unlike XACML, `AAL` defines a true negation (a permission is a negation of a prohibition, and it allows negative obligations), and explicit quantifiers. `AAL` enables subtype hierarchy for resources, subjects, roles, actions and provides attributes but also dynamic roles and roles sharing . The main strength of XACML is its policy enforcement. Our approach for enforcement is based on a monitoring technique which is presented in the next chapter.

XACML can manage these features by using an external component

The authors in [BCM14] advocate for strong accountability. The authors put forward strong accountability as a set of precise legal obligations supported by an effective software tool set. They demonstrate that the state of the art in terms of technology is sufficient to ensure the notion of accountability by design. Our work is a corner stone in the direction of strong accountability, an attempt to connect real accountability obligations with software design. There are theoretical formal models like [Ced+05]; [EW07]; [Jag+09] but they do not provide concrete languages and means to verify accountability at design time.

Another related work is [TDW13] which focuses on purpose restrictions governing information use. This paper provides a formal semantics for purpose restrictions which is based on simulating ignorance of

the prohibited information. Our notion of purpose is simply an information string associated with an action.

The most closely related work is [ZWL10], authors provide a formal service contract for accountable SaaS services. However, the decidability requirement, the contract language and the reasoning technique are different. It proposes a formal model, called OWL-SC, and a representation of the contracts based on ontology mixing two languages OWL-DL and SWRL. The paper also presents a translation of these contracts into colored Petri net. This allows to check properties and to reason on the contracts with Colored Petri nets (CNP) tools.

Several approaches to contract specification define specific modalities generally classified into permission or authorization, obligation, and prohibition. There are mainly three approaches: relying on deontic logic as in [PD11], using a pure temporal approach as we did, or a mix of both, for instance [BBF06]. Deontic logic appears as a natural choice. However it is subject to a few confusing paradoxes. Mixing both results in complex languages for which effective tool support is lacking. We rely on a uniform approach based on temporal operators and specific actions for permissions and prohibitions. This makes the result more understandable and we can reuse existing tools: provers or model-checkers.

As far as we know our work is the first providing a first-order linear temporal formula expressing accountability. But there are several points and open issues, starting with the usability of the language, even if `AAL` is close to natural language it still requires to have some technical background to use it. Recently we added the notion of templates in `AAL` which facilitates policy writing for the privacy officers, but we still need to develop more templates to cover more use-cases.

Concerning the semantics of `AAL`, a major limitation is the monodic condition that represents an expressiveness limitation, but we do not face this limitation in real use-cases. In the future we expect to relax the monodic constraint. For instance, `(always FORALL X,Y P(X, Y)) => FORALL X,Y always P(X, Y))` is a valid property. While the conclusion of the right-hand side is not monodic, it can be proved with the left-hand side. Also a true equality is missing in the language semantics.

4

ACCOUNTABILITY ENFORCEMENT

Contents

"I Never Think of the Future. It Comes Soon Enough."

— **Albert Einstein**

4.1 Introduction

Checking the consistency and compliance of policies is necessary but not sufficient, indeed we need to be able to enforce or to check that the real execution of a system matches with the specified behavior in the policy language. In order to achieve this goal, we can either enforce the policy by synthesizing a program or use runtime verification to monitor the system and report policy violations. In this chapter we first talk about the runtime verification of temporal logic properties and the problems introduced due to our application context, then we introduce a new logic called *FO-DTL*^{3} and we show how it answers our needs. Next we present the monitoring technique and the construction of *FO-DTL*^{3} monitors. Finally, we show the link between `AAL` and *FO-DTL*^{3} and how we can monitor `AAL` policies. The monitor implementation and the tools developed around this contribution are presented in Chapter 5.

4.2 From global to local specifications

When we design a system it's easier to specify the behavior of the system with a global vision, but when we move to the implementation phase we need to specify a local behavior and check that the composition of these behaviors respects the global behavior defined previously.

Enforcing an accountability policy has to do with runtime monitoring which is a topic discussed in several articles. For instance, [Sen+04] presents a temporal language with past operators and the synthesis of monitor in a distributed context. This approach proposes a temporal language able to express a local policy for distributed agents and then the translation of these policies into automata for their monitoring. Our purpose is slightly different since we are interested in enforcing accountability policies and not only to monitor safety properties.

This approach can be reused as a basis, for its language and synthesis algorithm. However, it seems really more difficult to directly translate our AAL sentences into distributed programs. One reason is that we have a logical approach with a global meaning. It means that we have two problems to solve: agent distribution and monitor synthesis.

Thus a pragmatic view is to consider first a projection step then a synthesis step. The projection step is responsible to transform one global `AAL` sentence involving several agents into a set of local `AAL` sentences (one for each agent). The synthesis step is to transform a local `AAL` sentence into a program or a monitor for enforcement of the local policy. To illustrate the projection, let us consider the global sentence in the following Listing:

```
FORALL d:Data ALWAYS
 IF (d.subject=Kim) THEN {
  (PERMIT Kim.read(d) AND PERMIT Kim.write(d) AND PERMIT Kim.input[
    CloudX](d)
  AND PERMIT CloudX.store(d) AND PERMIT CloudX.notify[Kim](d)
  AND DENY CloudX.write(d) AND PERMIT CloudX.read(d)
  AND (Kim.write(d) UNTIL Kim.input[CloudX](d))
  AND (IF CloudX.store(d) THEN CloudX.notify[Kim](d)))
 }

// Kim projection
FORALL D:Data ALWAYS
 IF (D.subject==Kim) THEN
  (PERMIT Kim.read(D) AND (PERMIT Kim.write(D) AND (PERMIT Kim.input[
    CloudX](D)
  AND ((Kim.write(D) U Kim.input_R[CloudX](D)) AND CloudX.notify_P[Kim
    ](D)))))
 }

// CloudX projection
FORALL D:Data ALWAYS
 IF (D.subject==Kim) THEN {
  (PERMIT CloudX.store(D) AND (PERMIT CloudX.notify[Kim](D)
  AND (DENY CloudX.write(D) AND (PERMIT CloudX.read(D) AND (F Kim.
    input_P[CloudX](D)
```

```
  AND (IF CloudX.store(D) THEN CloudX.notify_R[Kim](D)))))))
 }
```

We can produce the two local AAL sentences of the Listing above by projecting the original sentence to the agents Kim and CloudX. The new services with suffixes $_R$ and $_P$ represent the emission and the receipt associated with each global service interaction. It is possible to verify logically that the composition of the local AAL sentences, taking account the needed synchronizations, ensures the original global behavior.

Thus systematizing the above projection principle it seems possible to automatically project the global AAL sentences on each agent. Then automata synthesis from a local projection can be done using a technique similar to the one exposed in [Sen+04]. However, such an approach is only suitable with synchronous agents, that is agents have a common clock and there is a central mechanism to ensure communications and synchronization. But this is not adequate with distributed asynchronous agents evolving at their own speed and if the network latency or the criticality of some applications could impact the global system behavior. The problem of the behavioral synthesis in a true distributed context has been studied for a long time ago, especially in the context of web services choreography. The problem is complex (for instance, with synchronous communications is known not to be tractable) in general but various sufficient conditions to get it decidable have been established [KP06]; [GS12]; [McN10]; [RS14] among others. There are several issues to solve before implementing one of these solutions. The first issue is to choose adequate conditions for our context and to define the projection algorithm. Note that all the previous quoted work has been done in the context of linear temporal logic, thus generally we need to extend it to take into account first-order structures. The second issue is to extend the automata synthesis for *LTL* or Past Linear Temporal Logic (PLTL) to take into account our first-order logic.

In the following we will present how we resolve these issues using a logic called *FO-DTL*[3] .

4.3 Verification of temporal logic at runtime

In model checking and theorem proving we need to check all the possible executions of a system, whereas in runtime verification we only need to consider one execution sequence which is a finite prefix of a potentially infinite trace. Our goal is to be able to monitor AAL policies, thus we consider the following points:

1. Monitoring first order temporal logic: since the AAL language is based on first order temporal logic, we need to be able to monitor the first order case.
2. Logic over infinite traces: while standard model checking deals with infinite traces, runtime verification considers finite traces. Thus we need to mimic the *LTL* semantics over finite traces.

3. Distributed system architecture: in our context we deal with several actors which are distributed over a system, we need a means to handle this distribution.
4. Communications: in the real world, actors can communicate with each other using synchronous or asynchronous communications, thus our monitoring technique should be able to manage both cases.
5. Safety and liveness: safety properties describe that something bad should never happen (e.g $G(\neg bad)$), and liveness properties describes that something good will always happens (e.g $G(F(good))$).

For more details on runtime-verification the reader can refer to the surveys [GP10]; [BB13].

4.3.1 Monitoring the first order case

Several works exist in literature that tackle monitoring of first order logics. In [Cho95] authors present a technique to verify temporal constraints specified using metric past temporal logic. Basin et al. [BKM10] extend Chomicki's monitor with bounded future operators using the same logic. Authors in [HV08] present a technique to generate Buchi automaton on the fly for a logic with quantifications like in [BKV13]. There are several works that deal with parametric monitoring which offers support for monitoring traces carrying data [CR09]; [All+05]; [Sto10], even if it is not based on first-order logic. In [BKV13] authors propose monitoring technique for a first order logic in which quantified variables range over elements that occur at the current position of the trace.

4.3.2 Finite trace interpretation and Three-Valued-Logic

The semantic of *LTL* and *FOTL* are generally interpreted over infinite traces. As mentioned before, in the case of runtime verification a monitor has to check if the current trace is a prefix of an infinite trace. In this context we need to distinguish between the fact that a property is not yet satisfied, or if it is falsified. As mentioned in [BLS10], monitoring a property leads to three different states:

1. The property is satisfied after a finite number of steps.
2. The property is falsified for every possible continuation.
3. The observed prefix allows different continuations leading to either satisfaction or falsification.

This can be represented using a Three-Valued-Logic family, in which an evaluation leads to (`true`, `false` or `?`). This logic is a part of the many-valued logic family in which there are more than two truth values.

In [DGV13] authors present LDL^f a linear dynamic logic on finite traces which includes regular expressions, in order to overcome the

expressiveness limits of LTL^f over LTL on infinite traces. For example, the formula $\Diamond\Box\varphi$ that states in the future there exists a point where φ holds till the end. This is equivalent to $\Diamond(Last \wedge \varphi)$, i.e φ must hold at the last event of the trace, and this differs from the semantics. However in our case the use of Three-Valued-Logic solves the problem. Indeed, the previous formula will be evaluated to `?` if φ doesn't hold at the last point of the trace instead of `false`. We are able to mimic the semantic of LTL over infinite traces with finite traces. In [BLS10] the authors give a review of LTL-derived logics for finite traces from a runtime-verification perspective and they also present a technique to generate an automaton for LTL_3.

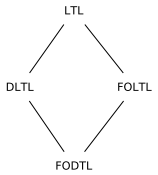
4.3.3 Dealing with distribution

Having several components to monitor in the context of a real distributed system, generally they communicate with each other with asynchronous communications and the notion of time differs for each component (i.e each component have his own local clock). The simplest approach is to have a central monitor that collects traces from other monitors and performs a computation to check the given formula. But exchanging messages for monitoring purpose only introduces an overhead in network communications which can be exponential in size in the number of events [Sen+04].

In [Sen+04] authors introduces *PT-DTL* (Past distributed temporal logic) which is an extension of *PLTL* (Past linear temporal logic). The benefits of this logic is that we can monitor a global property using local monitors at each actor of the system. The translation between the past and future is possible using `U` and `R` operators [LS95]. However this approach handles only safety properties. In [BF11] the authors present a decentralized LTL monitoring technique based on decentralized progression which needs to send messages for monitoring purpose only.

4.4 *FO-DTL³* : a first order temporal distributed logic

In [SS14] the authors tackle the runtime verification in distributed asynchronous systems. In their approach they extend $PTLTL$ [Sen+04] and LTL_3 [BLS11]. Here we introduce a logic called *FO-DTL³* (Three-Valued First Order Linear Distributed Temporal Logic), this logic is a mix between LTL^{FO} [BKV15] and DTL [Sen+04].



FO-DTL³ *logic inheritance diagram.*

4.4.1 The syntax of *FO-DTL³*

In *FO-DTL³* we find the classical propositional and LTL operators. We also have the operator @ called the location operator that allows distribution i.e a formula with the form $@_k\psi$ means that the formula

ψ is distributed on agent k. In addition to uninterpreted predicates $P(t^*)$ we allow interpreted predicates $I(t^*)$ and functions $f(\varphi^*)$. The syntax of *FO-DTL*^{3} is described in Table 9.

ψ	::=	$@_i\psi_i$	(`Top level formula`)
ψ_i	::=	`true` $\mid$ `false` $\mid \neg\psi_i \mid \psi_i(\vee \mid \wedge \mid \Rightarrow)\psi_i \mid \varphi$	(`propositional formulas`)
	$\mid$	$\bigcirc\psi_i \mid \Box\psi_i \mid \Diamond\psi_i \mid \psi_i \,\mathcal{U}\, \psi_i \mid \psi_i \,\mathcal{R}\, \psi_i$	(`temporal formulas (future)`)
	$\mid$	$\exists$`V`$.\psi_i \mid \forall$`V`$.\psi_i$	(`first-order formulas`)
	$\mid$	$@_k\psi_k$	(`distribution operator`)
φ_i	::=	$P(t^*) \mid I(t^*) \mid f(\varphi_i^*)$	(`predicates and functions`)
	$\mid$	`C` $\mid$ `V`	(`constants and variables`)

Table 9 – First Order Distributed Linear Temporal Logic

The top level *FO-DTL*^{3} formula $@_i\psi_i$ where i is an actor, is always defined on a local actor. For example, the property *"whenever a presence is detected by the sensor, the alarm should ring"* is expressed as follows:
in the point of view of the alarm:

$$@_{alarm}(@_{sensor}(detected) \Rightarrow ring)$$

or in the point of view of the sensor:

$$@_{sensor}(detected \Rightarrow @_{alarm}(ring))$$

Note that since we use local monitors (presented later in this chapter), we do not allow free variables outside localization operators. Let us consider the following example:

$$@_{alarm}(\forall x @_{sensor}(P(x)) \Rightarrow ring)$$

The formula $P(x) \Rightarrow ring$ is monitored locally, and the domain of the variable x is not known inside *sensor*. To do that, we have to synchronize the domains of quantified variables between each actor, which introduces a communication overhead.

4.4.2 Semantics of *FO-DTL*^{3}

FO-DTL^3 is interpreted over infinite traces. This logic extends LTL^{FO} that differs from *FOTL* in that the quantified variables range over the current element of the trace instead of the whole trace. Before we present the semantics of *FO-DTL*^{3} we introduce some concepts and definitions of *DTL* from [Sen+04]; [SS14].

Definition 8 (Last known position)**.** Let $a, b \in Agents$, $w \in Traces$:

$$last_a(w, b, h) = k$$

means that agent a at the position h of its execution knows that k is the last known position of agent b on its execution.

Definition 9 (Known prefix)**.** Let $a, b \in Agents$, $w, u \in Traces$:

$$known_b(w,h) = (u^{a_0}, u^{a_1}, \dots u^{a_n})$$
$$\text{such as } \forall a \in Agents, u^a = w^a_{0..k} \text{ if } k = last_b(w,a,h) \text{ is defined.}$$

Which means that agent b at the position h of its execution knows a prefix of the execution trace u^a of all agents a where the prefix size is k which is the last known position of agent a by b.

Let $i \in Agents$, π_i is the execution trace of agent i and p a position in the trace π_i, the semantic of a local *FO-DTL*³ formula is defined as follows:

$$\begin{aligned}
(\pi_i,p) \vDash \texttt{true} &\overset{\text{def}}{\Leftrightarrow} \texttt{true}\\
(\pi_i,p) \vDash \texttt{false} &\overset{\text{def}}{\Leftrightarrow} \texttt{false}\\
(\pi_i,p) \vDash \neg\psi &\overset{\text{def}}{\Leftrightarrow} \neg((\pi_i,p) \vDash \psi)\\
(\pi_i,p) \vDash (\psi_1 \vee \psi_2) &\overset{\text{def}}{\Leftrightarrow} (\pi_i,p) \vDash \psi_1 \vee (\pi_i,p) \vDash \psi_2\\
(\pi_i,p) \vDash (\psi_1 \wedge \psi_2) &\overset{\text{def}}{\Leftrightarrow} (\pi_i,p) \vDash \psi_1 \wedge (\pi_i,p) \vDash \psi_2\\
(\pi_i,p) \vDash \varphi &\overset{\text{def}}{\Leftrightarrow} \varphi(\pi_i(p))\\
(\pi_i,p) \vDash \exists x.\psi &\overset{\text{def}}{\Leftrightarrow} \exists x \in U.((\pi_i,p) \vDash \psi)\\
(\pi_i,p) \vDash \forall x.\psi &\overset{\text{def}}{\Leftrightarrow} \forall x \in U.((\pi_i,p) \vDash \psi)\\
(\pi_i,p) \vDash \texttt{X}\psi &\overset{\text{def}}{\Leftrightarrow} ((p+1) < |\pi_i|) \wedge ((\pi_i,p+1) \vDash \psi)\\
(\pi_i,p) \vDash \psi_1 \texttt{U} \psi_2 &\overset{\text{def}}{\Leftrightarrow} \exists q.(p \leq q < |\pi_i|) \wedge ((\pi_i,q) \vDash \psi_2) \wedge (\forall q'.(p \leq q' < q) \Rightarrow\\
&\quad ((\pi_i,q') \vDash \psi_1))\\
(\pi_i,p) \vDash \psi_1 \texttt{R} \psi_2 &\overset{\text{def}}{\Leftrightarrow} (\forall q.(p \leq q < |\pi_i|) \Rightarrow ((\pi_i,q) \vDash \psi_2)) \vee \exists q.(p \leq q < |\pi_i|)\\
&\quad \wedge((\pi_i,q) \vDash \psi_1) \wedge (\forall q'.(p \leq q' < q) \Rightarrow ((\pi_i,q') \vDash \psi_2))\\
(\pi_i,p) \vDash \texttt{G}\psi &\overset{\text{def}}{\Leftrightarrow} \forall q.(p \leq q < |\pi_i|) \Rightarrow ((\pi_i,q) \vDash \psi)\\
(\pi,p) \vDash \texttt{F}\psi &\overset{\text{def}}{\Leftrightarrow} \exists q.(p \leq q < |\pi_i|) \wedge ((\pi_i,q) \vDash \psi)\\
(\pi_i,p) \vDash @_x\psi &\overset{\text{def}}{\Leftrightarrow} (\pi_x,k) \vDash \psi \ \texttt{where}\ k = last_i(\pi_i,x,p) \overset{\text{def}}{\Leftrightarrow} known_i(\pi_i,p) \vDash \psi
\end{aligned}$$

Table 10 – The semantics of *FO-DTL*³

4.5 *FO-DTL*³ monitoring

Overview. We recall that in our context a system is composed of several agents that communicate with each other by exchanging messages. We call an actor, an agent in the system with a reference monitor attached to it.

Definition 10 (System actor)**.** An actor a of the system is defined as follows:
RM_a=$\{agent, \psi, w, M, \Sigma, \theta\}$ where:

- agent is the name of monitor's actor;
- ψ the main *FO-DTL*^{3} formula to monitor;
- w a infinite trace;
- M the main reference monitor;
- Σ sub formulas from other actors to monitor;
- θ a set of {SEND/RECEIVE/INTERNAL} events that correspond respectively to (emitted, received, internal) messages by the actor.

For each agent i in the system we attach a reference monitor RM_i, which monitors a formula ψ_i. We extract the sub formulas of ψ_i that are inside a localization operator @ and we distribute these formulas on the corresponding remote actor. More formally we have the definition that follows (the expression $@^i_j\psi_x$ denotes a remote formula ψ_x specified by actor i on actor j.):

$$\forall i \in agents. \forall @^i_j\psi_{jn} \in \varphi_i \texttt{ where } \varphi_i = \{@^i_j\psi_{j0}, @^i_j\psi_{j1}, \dots @^i_k\psi_{kn}\} : RM_j[\Sigma] = RM_j[\Sigma] \cup \{\psi_{jn}\}$$

Let us consider a simple ping-pong example to illustrate the monitoring principle: we have two agents `bob` and `alice` that communicates with each other using actions `send, reply`. `bob` wants to monitor the property that states: if `bob` sends a ping message to `alice`, then `alice` should reply to him with a pong message.

$$@_{bob}(F(send(bob, alice, ping) \Rightarrow F(@_{alice}(reply(alice, bob, pong))))$$

Generating the monitors for this formula gives the following configuration:

1. on actor **bob**: one monitor that monitors the main formula: $F(send(bob, alice, ping) \Rightarrow F(@_{alice}(reply(alice, bob, pong))))$
2. on actor **alice**: one sub monitor that monitors the formula: $reply(alice, bob, pong)$ for bob.

Details on how actors exchange information will be discussed in the next sections.

4.5.1 Monitors construction

In the following we present the technique to monitor *FO-DTL*^{3} formula. First we explain the principle of the progression technique [BK98]; [Zha+10].

Concept. We use the formula progression technique also called rewriting technique, the principle is to split a given formula into two formulas, the first expresses what needs to be satisfied at the current event and the second one expresses what needs to be satisfied in the next events (future).

From classical semantics to progression. Starting from the classical semantics of LTL_3, we want to express inductively an LTL_3 formula at state n depending on an LTL_3 formula at state n+1. For example, $\mathtt{G}(\psi)_{\pi,n}$ is expressed inductively by $\psi_{\pi,n} \wedge \mathtt{G}(\psi)_{\pi,n+1}$, where $\psi_{\pi,n}$ is the formula that we need to satisfy at the current step and $\mathtt{G}(\psi)_{\pi,n+1}$ is the formula that we need to satisfy in the future. Below the proof of this transformation:

Note that these two notations $\mathtt{G}(\psi)_{\pi,n}$ and $(\pi,n) \vDash \mathtt{G}\psi$ are equivalent.

Proof. $(\pi,n) \vDash \mathtt{G}\psi \overset{\text{def}}{\Leftrightarrow} \forall q.(n \leq q < |\pi|) \Rightarrow ((\pi,q) \vDash \psi) \Leftrightarrow$
we split the $\leq$ *into* $<$ *and* $=$
$\forall q.(n < q < |\pi|) \Rightarrow ((\pi,q) \vDash \psi) \wedge \forall q.(n = q < |\pi|) \Rightarrow ((\pi,n) \vDash \psi) \Leftrightarrow$
we use the equivalence $n < q \equiv n+1 \leq q$
$\forall q.(n+1 \leq q < |\pi|) \Rightarrow ((\pi,q) \vDash \psi) \wedge ((\pi,n) \vDash \psi) \Leftrightarrow$
$(\pi,n+1) \vDash \mathtt{G}\psi \wedge ((\pi,n) \vDash \psi)$

Following the same principle we derive the new semantics for the five LTL operators (for the other operators the semantic form does not change)

$$
\begin{aligned}
(\mathtt{X}\psi)_{\pi,n} &\overset{\text{def}}{\Leftrightarrow} \psi_{\pi,n+1} \\
(\psi_1 \ \mathtt{U} \ \psi_2)_{\pi,n} &\overset{\text{def}}{\Leftrightarrow} \psi_{2\pi,n} \vee (\psi_{1\pi,n} \wedge (\psi_1 \ \mathtt{U} \ \psi_2)_{\pi,n+1}) \\
(\psi_1 \ \mathtt{R} \ \psi_2)_{\pi,n} &\overset{\text{def}}{\Leftrightarrow} \psi_{1\pi,n} \vee (\psi_{2\pi,n} \wedge (\psi_1 \ \mathtt{R} \ \psi_2)_{\pi,n+1}) \\
(\mathtt{G}\psi)_{\pi,n} &\overset{\text{def}}{\Leftrightarrow} \psi_{\pi,n} \wedge \mathtt{G}(\psi)_{\pi,n+1} \\
(\mathtt{F}\psi)_{\pi,n} &\overset{\text{def}}{\Leftrightarrow} \psi_{\pi,n} \vee \mathtt{F}(\psi)_{\pi,n+1}
\end{aligned}
$$

Table 11 – The progression semantics of LTL operators

Proof.
Next operator.
$(\pi,n) \vDash \mathtt{X}\psi \overset{\text{def}}{\Leftrightarrow} ((n+1) < |\pi|) \wedge ((\pi,n+1) \vDash \psi) \Leftrightarrow (\pi,n+1) \vDash \psi$

Future operator.
$(\pi,n) \vDash \mathtt{F}\psi \overset{\text{def}}{\Leftrightarrow} \exists q.(n \leq q < |\pi|) \Rightarrow ((\pi,q) \vDash \psi) \Leftrightarrow$
$\exists q.(n < q < |\pi|) \Rightarrow ((\pi,q) \vDash \psi) \vee \exists q.(n = q < |\pi|) \Rightarrow ((\pi,n) \vDash \psi) \Leftrightarrow$
$\exists q.(n+1 \leq q < |\pi|) \Rightarrow ((\pi,q) \vDash \psi) \vee ((\pi,n) \vDash \psi) \Leftrightarrow$
$(\pi,n+1) \vDash \mathtt{F}\psi \vee ((\pi,n) \vDash \psi)$

Until operator.
$(\pi,n) \vDash \psi_1 \mathtt{U} \ \psi_2 \overset{\text{def}}{\Leftrightarrow} \exists q.(n \leq q < |\pi|) \wedge ((\pi,q) \vDash \psi_2) \wedge$
$\forall q'.(n \leq q' < q) \Rightarrow ((\pi,q') \vDash \psi_1) \Leftrightarrow$
$(\exists q.(n = q < |\pi|) \wedge ((\pi,n) \vDash \psi_2) \vee \exists q.(n < q < |\pi|) \wedge ((\pi,q) \vDash \psi_2)) \wedge$
$(\forall q'.(n = q' < q) \Rightarrow ((\pi,n) \vDash \psi_1) \wedge \forall q'.(n < q' < q) \Rightarrow ((\pi,q') \vDash \psi_1) \Leftrightarrow$
$(\exists q.(n = q < |\pi|) \wedge ((\pi,n) \vDash \psi_2) \vee \exists q.(n+1 \leq q < |\pi|) \wedge ((\pi,q) \vDash$

$\psi_2)) \wedge$
$(\forall q'.(n = q' < q) \Rightarrow ((\pi, n) \vDash \psi_1) \wedge \forall q'.(n + 1 \leq q' < q) \Rightarrow ((\pi, q') \vDash \psi_1))$
$\Leftrightarrow$ *we apply the distributive law of conjunction*
$(\exists q.(n = q < |\pi|) \wedge ((\pi, n) \vDash \psi_2)) \wedge (\forall q'.(n = q' < q) \Rightarrow ((\pi, n) \vDash \psi_1) \wedge \forall q'.(n + 1 \leq q' < q) \Rightarrow ((\pi, q') \vDash \psi_1) \vee$
$(\exists q.(n + 1 \leq q < |\pi|) \wedge ((\pi, q) \vDash \psi_2)) \wedge (\forall q'.(n = q' < q) \Rightarrow ((\pi, n) \vDash \psi_1 \wedge$
$\forall q'.(n + 1 \leq q' < q) \Rightarrow ((\pi, q') \vDash \psi_1)$
$\Leftrightarrow$ *we apply the associative law of conjunction*
$((\pi, n) \vDash \psi_2) \vee e(((\pi, n) \vDash \psi_1) \wedge (\pi, n + 1) \vDash \psi_1 \mathtt{U}\ \psi_2)$

Release operator. *The proof can be derived directly from the Until proof using the equivalence* $\psi_1 \mathtt{R} \psi_2 \equiv \neg(\neg\psi_1 \mathtt{U} \neg\psi_2)$
$(\pi, n) \vDash \psi_1 \mathtt{R}\ \psi_2 \overset{\mathtt{def}}{\Leftrightarrow} ((\pi, n) \vDash \psi_1) \vee (((\pi, n) \vDash \psi_2) \wedge (\pi, n + 1) \vDash \psi_1 \mathtt{R}\ \psi_2)$

Below we present the general progression function.

Definition 11 (LTL_3 progression function)**.** Let $\psi, \psi_1, \psi_2 \in LTL_3$ and an event $\sigma \in \Sigma$. The progression function $P_{ltl} : LTL_3 \times \Sigma \to LTL_3$ is inductively defined as follows:

$$
\begin{array}{rcl}
P_{ltl}(\top, \sigma) & = & \top \\
P_{ltl}(\bot, \sigma) & = & \bot \\
P_{ltl}(p \in \mathrm{AP}, \sigma) & = & \top, \text{if } p \in \sigma, \bot \textit{ otherwise} \\
P_{ltl}(\neg\psi, \sigma) & = & \neg_3\ P_{ltl}(\psi, \sigma) \\
P_{ltl}(\psi_1 \vee \psi_2, \sigma) & = & P_{ltl}(\psi_1, \sigma) \vee_3 P_{ltl}(\psi_2, \sigma) \\
P_{ltl}(\psi_1 \wedge \psi_2, \sigma) & = & P_{ltl}(\psi_1, \sigma) \wedge_3 P_{ltl}(\psi_2, \sigma) \\
P_{ltl}(\mathtt{G}\ \psi, \sigma) & = & P_{ltl}(\psi, \sigma) \wedge_3 \mathtt{G}(\psi) \\
P_{ltl}(\mathtt{F}\ \psi, \sigma) & = & P_{ltl}(\psi, \sigma) \vee_3 \mathtt{F}(\psi) \\
P_{ltl}(\psi_1\ \mathtt{U}\ \psi_2, \sigma) & = & P_{ltl}(\psi_2, \sigma) \vee_3 (P_{ltl}(\psi_1, \sigma) \wedge_3 \psi_1\ \mathtt{U}\ \psi_2) \\
P_{ltl}(\psi_1\ \mathtt{R}\ \psi_2, \sigma) & = & P_{ltl}(\psi_1, \sigma) \vee_3 (P_{ltl}(\psi_2, \sigma) \wedge_3 \psi_1\ \mathtt{R}\ \psi_2) \\
P_{ltl}(\mathtt{X}\ \psi, \sigma) & = & \psi
\end{array}
$$

Table 12 – Progression algorithm for LTL_3

The truth tables of the operators $\wedge_3, \vee_3, \neg_3$ are defined below:

$\psi_1 \wedge_3 \psi_2$	$\top$	$\bot$	_
$\top$	$\top$	$\bot$	ψ_2
$\bot$	$\bot$	$\bot$	$\bot$
_	ψ_1	$\bot$	$\psi_1 \wedge \psi_2$

$\psi_1 \vee_3 \psi_2$	$\top$	$\bot$	_
$\top$	$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$	ψ_2
_	$\top$	ψ_1	$\psi_1 \vee \psi_2$

ψ	$\neg_3\psi$
$\top$	$\bot$
$\bot$	$\top$
_	$\neg\psi$

In addition we define a reduction function $eval_3$ (which is derived from LTL_3 semantics) that is applied on the rewritten formula:

$$eval_3(\psi) = \begin{cases} \top & if\ \psi = \top \\ \bot & if\ \psi = \bot \\ ? & otherwise \end{cases}$$

Example 10. Let us consider the formula $\psi = F(p \vee F(q))$ and the prefix $u = (\{a,b\}\ \{q\}\ \{b\})$ of an infinite execution trace w.

1. At the first step, we have $\psi = F(p \vee F(q))$ and the event $\{a,b\}$. We apply the progression function:
   $P_{ltl}(\psi, \{a,b\}) = P_{ltl}(p \vee_3 F(q)) \vee_3 F(p \vee_3 F(q)) = F(q) \vee F(p \vee_3 F(q))$ and $eval_3(F(q) \vee F(p \vee_3 F(q)))$ gives ?.
2. At the second step, the new rewritten formula is now:
   $\psi = F(q) \vee F(p \vee_3 F(q))$
   we apply the progression function:
   $P_{ltl}(\psi, \{q\}) = \underbrace{\underbrace{P_{ltl}(F(q))}_{true \vee_3 F(q)} \vee_3 \underbrace{P_{ltl}(F(p \vee_3 F(q)))}_{p \vee_3 q \vee_3 F(q) \vee_3 F(p \vee_3 F(q))}}_{true}$
   and $eval_3(true)$ is $\top$.
3. At the third step, the new rewritten formula is *true* which always gives $\top$, so there is no need to continue the progression.

4.5.2 First order and distribution extensions

4.5.2.1 Monitoring FOTL

The first order introduces variables, interpreted predicates, functions and two operator $\forall$ and $\exists$. Note that the progression function does not consider the existential quantifier, since it is expressed with the universal quantifier with the equivalence relation: $\exists x.\psi \equiv \neg(\forall x.\neg\psi)$. We recall that this extension is based on [BKV15] which imposes the restriction that the quantified variables range over the current element of the trace instead of the whole trace.

Definition 12 (*FOTL progression function*)**.** Let $\psi \in FOTL_3$, an event $\sigma \in \Sigma$ and $\xi_{x_i \leftarrow v_i}$ a valuation that associates the value v_i to the

variable x_i. The progression function for a *FOTL* formula, is obtained by extending P_{ltl} with the rule that follows:

$$\begin{aligned} P_{fotl}(\forall x.\psi, \sigma) &= P_{fotl}(\wedge_3^{i=0..n}(\psi, \xi_{x_i \leftarrow v_i}), \sigma) \\ P_{fotl}(\wedge_3^{i=0..n}(\psi, \xi_{x_i \leftarrow v_i}), \sigma) &= \wedge_3^{i=0..n} P_{fotl}((\psi, \xi_{x_i \leftarrow v_i}), \sigma) \end{aligned}$$

A universal quantified formula $\forall x.\psi$ is rewritten as a conjunction of formula ψ with an evaluation for each x that appears in the event σ. The conjunction of valued formulas is rewritten by a conjunction of the progression for each valued formula. The interpreted predicates and functions are computable thus they do not appear in the progression function.

The truth table of $\wedge_3^{i=0..n}(\psi_i)$ is as follows:

$$\wedge_3^{i=0..n}(\psi_i) = \begin{cases} \top & if\ \forall i = 0..n\ .\ \psi_i = \top \\ \bot & if\ \exists i = 0..n\ .\ \psi_i = \bot \\ \wedge_3^{i=0..n}(\psi_i) & otherwise \end{cases}$$

Example 11. Let us consider the formula $\psi = G(\forall x : T.\ P(x))$ and the prefix
$u = (\{\mathrm{P(b)}\}\ \{\mathrm{T(a)},\ \mathrm{P(a)}\}\ \{\mathrm{T(b)}\})$ of an infinite execution trace w.

1. At the first step, $\psi = G(\forall x : T.\ P(x))$
   we apply the progression function:
   $P_{fotl}(\psi, \{P(b)\}) = \underbrace{\underbrace{P_{fotl}(\forall x : T.\ P(x))}_{\wedge_3^{i=0..0}(\varnothing)} \wedge_3 G(\forall x : T.\ P(x))}_{true\ \wedge_3\ G(\forall x:T.\ P(x)) \equiv G(\forall x:T.\ P(x))}$
   and $eval_3(G(\forall x : T.\ P(x))) = ?$.
2. In the second step, the rewritten formula remains the same as the original, but this time the valued formula conjunction that results from the progression changes since we have $T(a)$ in the current event.
   $P_{fotl}(\psi, \{T(a), P(a)\}) = \underbrace{\underbrace{P_{fotl}(\forall x : T.\ P(x))}_{\wedge_3^{i=0..1}(P(a))} \wedge_3 G(\forall x : T.\ P(x))}_{true\ \wedge_3\ G(\forall x:T.\ P(x)) \equiv G(\forall x:T.\ P(x))}$

   Thus, the result remains the same because $\wedge_3^{i=0..1}(P(a))$ is evaluated to $\top$ which gives the same formula as in step 1, and $eval_3(G(\forall x : T.\ P(x)))$ gives ?.
3. The third step is similar to step2, but here $\wedge_3^{i=0..1}(P(b))$ is evaluated to $\bot$ and this is because we check $P(b)$ on the current event only, if we consider the whole trace the result would be $\top$ since $P(b)$ appears in the first event of the trace.

$$P_{fotl}(\psi, \{T(b)\}) = \underbrace{\underbrace{P_{fotl}(\forall x : T.\ P(x))}_{\wedge_3^{i=0..1}(P(b))} \wedge_3 G(\forall x : T.\ P(x))}_{False \wedge_3 G(\forall x{:}T.\ P(x)) \equiv False}$$

and $eval_3(False)$ is $\bot$.

4.5.2.2 Monitoring DLTL

When we deal with distributed systems we face to a well-known issue which is the causality preservation. As mentioned before, a common solution is the vector clock algorithm. Here we use the knowledge vectors inspired from [Sen+04] which are based on vector clocks.

Definition 13 (knowledge vector)**.** Let $KV_i[x_\psi]$ denote an entry for the remote formula ψ of agent x in the knowledge vector KV of agent i. An entry $KV_i[x_\psi]$ contains a value (val) that represents the evaluation of the remote monitor on x for the formula ψ which can be $(\top, \bot, ?)$, and a sequence (seq) which represents the current step of the remote monitor. Each actor updates its knowledge vector using the following rules:

— *INTERNAL*: $\forall \psi_j \in \Sigma.\ KV_i[x_{\psi_j}].val = eval_3(\psi_j)$
— *SEND msg to* x: $KV_i[x_{\psi_j}].val = eval_3(\psi_j)$
— *RECEIVE msg from* x:
$\forall j\ if\ KV_m[j].seq > KV_i[j].seq\ then\ KV_i[j] \longleftarrow KV_m[j]$

We notice that the knowledge vectors are updated only if a message is exchanged between actors, thus we do not introduce specific messages used for monitoring purposes only, in order to avoid a communication overhead in the network. Other solutions exist, for example we can use a smart broadcasting by sending KV updates to other actors only when the result changes (from ? to $\bot$ or $\top$).

Definition 14 (DTL *progression function*)**.** The progression function for a DTL formula, is obtained by extending P_{ltl} with the rule that follows:

$$P_{dtl}(@_x(\psi), \sigma) \quad = \quad KV[x_\psi].val$$

Example 12. We take the ping-pong example presented previously with the traces: u_{bob} = ({send(bob,alice,ping)} {q} {p}) u_{alice} = ({q} {reply(alice,bob,pong)} {p}).

$$@_{bob}(F(send(bob, alice, ping) \Rightarrow F(@_{alice}(reply(alice, bob, pong))))$$

The figure below illustrates the DTL monitoring of this example.

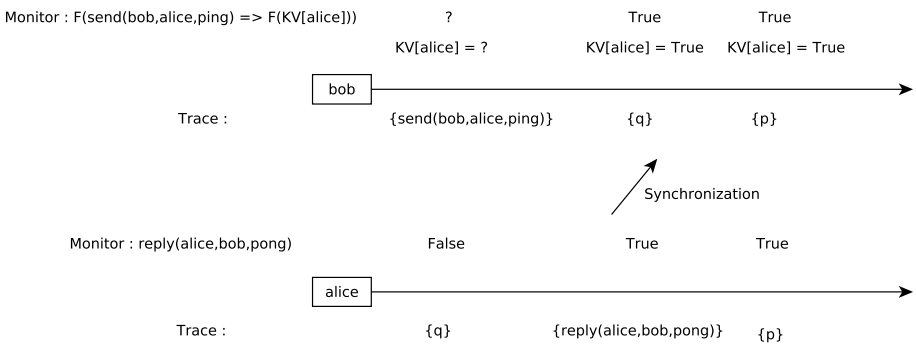


Figure 3 – DTL monitoring example

Definition 15 (FO-DTL^3 *progression function*)**.** The progression function for an *FO-DTL^3* formula is obtained by combining P_{dtl} and P_{fotl}.

4.5.3 Monitor optimizations

The progression technique has the advantage to be close to the semantic and it is more flexible than classical monitoring techniques since the formula to monitor can be modified at runtime without any overhead, which is useful when we monitor accountability policies i.e. the monitored property can change if a violation occurs.

The disadvantage of the rewriting technique is that the progressed formula can diverge and grow in size depending on the number of events. For example, consider the formula $G(a \wedge F(b))$ with the trace $a, a, a, \ldots$. Applying the progression technique for 3 steps gives the following rewritten formula: $F(b) \wedge (F(b) \wedge (F(b) \wedge F(b) \wedge G(a \wedge F(b))))$. However as mentioned in [BF12], the addition of some practical simplification rules to the progression function usually prevents this problem from occurring. In [She+14] authors used Fix-point reduction in order to limit the size of the generated formula by progression. Formulas that contains the operators U, R and G, F by extension are more likely susceptible to lead to this problem, due to the expansion formula of these operators. The principle of the Fix-point reduction of [She+14] is as follows:

Theorem 4.1 (Fix-point Reduction)**.** *Let $f(1)$ and $f(2)$ formulas defined recursively as follows:*

$$f_1(1) = \psi_3 \vee (\psi_2 \wedge \psi_1) \text{ and } f_1(n) = \psi_{2n+1} \vee (\psi_{2n} \wedge f_1(n-1))$$
$$f_2(1) = \psi_3 \wedge (\psi_2 \vee \psi_1) \text{ and } f_2(n) = \psi_{2n+1} \wedge (\psi_{2n} \vee f_1(n-1))$$

When $n > 1$ and $n > k$ then:

1. $\psi_{2n+1} = \psi_{2k+1} \Rightarrow f_1(n) \equiv f_1(k)[\bot / \psi_{2k+1}]$
2. $\psi_{2n} = \psi_{2k} \Rightarrow f_1(n) \equiv f_1(k)[\top / \psi_{2k}]$
3. $\psi_{2n+1} = \psi_{2k+1} \Rightarrow f_2(n) \equiv f_2(k)[\top / \psi_{2k+1}]$

4. $\psi_{2n} = \psi_{2k} \Rightarrow f_2(n) \equiv f_2(k)[\bot\ /\psi_{2k}]$

Note that the progression formula of U and R operators, can be written in $f(1)$ and $f(2)$ forms :

$$\psi_1 U \psi_2 \equiv \psi_2 \vee \psi_1 \wedge X(\psi_1 U \psi_2)$$
$$\psi_1 R \psi_2 \equiv \psi_2 \wedge \psi_1 \vee X(\psi_1 U \psi_2)$$

Following the same principle, the progression of the liveness formula can be written in $f(1)$ form : $G(F(\psi)) \equiv \psi \vee F(\psi) \wedge X(G(F(\psi)))$

Another drawback is that the rewriting technique doesn't consider the semantic level. For example, consider the formula $F(a \wedge \neg a)$, an optimized automaton based monitor can easily detect that the formula is falsified, which is not the case of a monitor that uses the progression technique since it operates on the syntactic level only. Here again, we can introduce some simplification rules based on the semantics (which also reduce the written formula). Below we present an extended version of the simplification rules presented in [She+14].

1. $\psi \wedge \neg\psi \equiv \bot$
2. $(\psi_1 U \psi_2) \wedge \psi_2 \equiv \psi_2$
3. $(\psi_1 U \psi_2) \vee \psi_2 \equiv \psi_1 U \psi_2$
4. $(\psi_1 R \psi_2) \wedge \psi_2 \equiv \psi_1 R \psi_2$
5. $(\psi_1 R \psi_2) \vee \psi_2 \equiv \psi_2$
6. $(\psi_1 R \psi_2) \wedge \psi_1 \equiv \psi_1 \wedge \psi_2$
7. $(\psi_1 U \psi_2) \vee (\psi_3 U \neg\psi_2) \equiv \top$
8. $(\psi_1 R \psi_2) \wedge (\psi_3 R \neg\psi_2) \equiv \bot$

4.5.4 Monitor completeness and soundness

The progression formula is not complete [BLS10] thus we need to combine it with a theorem prover. We experimented this approach using the *TSpass* prover, even if it is more complete this approach is less effective and time consuming.

Let's consider the formula $\psi = G(p \vee \neg p)$ with an infinite trace, independently from the events in the trace, the formula will progress indefinitely: $true \wedge_3 true \wedge_3 \cdots \wedge_3 G(p \vee \neg p)$. We can prove co-inductively that the previous formula is valid. The same behavior for the formula $\psi = F(p \wedge \neg p)$ which progresses indefinitely: $false \vee_3 false \vee_3 \cdots \vee_3 F(p \wedge \neg p)$ we can prove inductively that this formula is not satisfiable.

Thus, we propose the following changes to the initial progression function:

$$P_{ltl}(\texttt{G}\ \psi, \sigma) = \begin{cases} \top & if\ \psi\ \texttt{is valid} \\ \bot & if\ \psi\ \texttt{is unsatisfiable} \\ P_{ltl}(\psi, \sigma) \wedge_3 \texttt{G}(\psi) & otherwise \end{cases}$$

$$P_{ltl}(\texttt{F}\ \psi, \sigma) = \begin{cases} \top & if\ \psi\ \texttt{is valid} \\ \bot & if\ \psi\ \texttt{is unsatisfiable} \\ P_{ltl}(\psi, \sigma) \vee_3 \texttt{F}(\psi) & otherwise \end{cases}$$

$$P_{ltl}(\psi_1 \texttt{ U } \psi_2, \sigma) = \begin{cases} \top & if\ \psi_2 \texttt{ is valid} \\ \bot & if\ \psi_2 \texttt{ is unsatisfiable} \\ P_{ltl}(\psi_2, \sigma) \vee_3 (P_{ltl}(\psi_1, \sigma) \wedge_3 \psi_1 \texttt{ U } \psi_2) & otherwise \end{cases}$$

$$P_{ltl}(\psi_1 \texttt{ R } \psi_2, \sigma) = \begin{cases} \top & if\ \psi_1 \texttt{ is valid} \\ \bot & if\ \psi_1 \texttt{ is unsatisfiable} \\ P_{ltl}(\psi_1, \sigma) \vee_3 (P_{ltl}(\psi_2, \sigma) \wedge_3 \psi_1 \texttt{ R } \psi_2) & otherwise \end{cases}$$

4.6 Monitoring AAL policies using *FO-DTL*^{3} .

We consider a ping-pong example, in AAL we have the following global behavior:

```
ALWAYS ( IF(bob.send[alice]("ping")) THEN { MUST alice.reply[bob]("pong
    ")} )
```

We want to monitor the global property that says: whenever bob sends a ping message to alice, then alice replies to him with a pong message. First we translate the AAL formula into an *FO-DTL*^{3} formula, the translation is similar to FOTL, except that we prefix the messages with the localization operator @, which gives us the following formula:

$$G(@_{bob}(send(bob, alice, ping)) \Rightarrow F(@_{alice}(reply(alice, bob, pong))))$$

We choose for example to project the property on bob's local monitor, and we produce three monitors (one for each @ operator and a local monitor for bob):

— Sub monitor 1 on bob: $send(bob, alice, ping)$
— Sub monitor 2 on alice: $reply(alice, bob, pong)$
— Main monitor on on bob: $G(KV[1] \Rightarrow F(KV[2]))$

The actors are synchronized when they communicate with each other.

4.7 Conclusion

In this chapter we introduced a new logic called *FO-DTL*^{3} which is a distributed first order temporal logic. We also presented a monitoring technique based on rewriting formula.

In [Sen+06] authors present a specific temporal logic *MTTL*, to express properties of asynchronous multi-threaded systems. Its monitoring procedure takes as input a safety formula and a partially ordered execution of a parallel asynchronous system. Then it determines if runs exist or not in the execution that violate the formula. Their monitors are restricted to safety formulae but in our case we want to be able to monitor liveness formulae (which are the most used in our use case). Another difference is that they use a global trace and in our case we assume that the system is not always able to collect a global trace.

Other works like [GMM06] target distributed systems but do not focus on the communication overhead that may be induced by the monitoring. In [SS14] authors tackles the runtime verification in distributed asynchronous systems. In their approach they extend *PTLTL* [Sen+04] and *LTL3* [BLS11]. They monitor *LTL* formula over a distributed system using a Bushi automaton. In our approach, we monitor formulas with the first order using the progression technique. However, we face some limitations: we do not allow free variables outside localization operators, otherwise we need to synchronize the domains of quantified variables between each actor, which introduces a communication overhead. Also the domain of quantified variables is limited to the current event only, we do not consider the full trace from the beginning. The synchronization of knowledge vectors only when actors communicate with each other may introduce a small delay on monitoring results. This can be fixed by forcing reference monitors to synchronize periodically but it introduces a communication overhead. This overhead can be reduced using heuristic analysis. Expressing `AAL` policies using *FO-DTL*^{3} is relatively simple, since in `AAL` we describe abstract behaviors. The translation will be more difficult in the case of more complex systems.

Finally, as seen before, the progression technique is not complete and needs to be mixed with a theorem prover. Currently, we work on a co-inductive monitoring technique in order to deal with this issue.

5

ACCOUNTABILITY TOOLS

Contents

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

— **Brian Kernighan.**

5.1 Introduction

In this chapter we present different tools that were developed in the context of this thesis. This set of tools represents an end-to-end accountability framework that helps to consider accountability in modern systems from the specification to the implementation. First we present the accountability laboratory `AccLab` which helps users to design a consistent set of policies for a specific system, using the abstract accountability component design and `AAL` language. Second we present `FodtlMon` which is the implementation of the monitoring framework for *FO-DTL*^{3} presented in Chapter 4. This tool is the core for two other accountability tools: the first one is the `AccMon` framework (Accountability monitoring) that we present in Section 5.4. This framework allows to monitor policies written in *FO-DTL*^{3} on real systems (web applications, hardware, operating systems, etc). The second tool that we present in Section 5.5 is `PyMon` which is a monitoring system for Python. It allows to specify different properties on monitor them on Python code and enforce Python type system with runtime verification.

5.2 AccLab: an accountability laboratory

The idea behind AccLab was born at the beginning of this thesis during a discussion with my director. At this time the language AAL was just a prototype with a very minimal grammar. We wanted to have a kind of sandbox or laboratory that allows us to develop policies and see accountability in action, and thus we came up with the name AccLab for Accountability Laboratory. The goal of AccLab framework is to handle accountability from design time and to guide the developers in the process of software development.

The first prototype of AccLab was developed since February 2014, the back-end was written in Java, and the front-end in PHP/JavaScript based on CODIAD (codiad.com) which is an open source web-based IDE. The framework was reimplemented from scratch in August 2014 and was released on github (github.com/hkff/AccLab) in February 2015 under GPL3 license. The last release of AccLab is version 2.2 which was released on 23 July 2017. The AccLab has 4870 Source lines of code (sloc) for the core and 6800 sloc for the UI. Now the back-end is written in Python3 and the front-end in JavaScript based on dockspawn (dockspawn.com) which is a web- based dock layout engine released under MIT license.

5.2.1 Framework presentation

AccLab workflow. We first introduce the workflow in AccLab illustrated by Figure 4. (1) First the system architect designs the system with all the actors and their communications using the abstract component design; (2) next, the privacy officer writes accountability policies using the abstract accountability language AAL; (3) with the support of an expert, the privacy officer checks the consistency and the compliance of the AAL policies and modifies the AAL clauses in order to obtain a coherent set of policies; (4) after that, they can perform simulations to see if everything works as expected; (5) finally,they can generate machine readable policies or monitoring specifications with the assistance of a developer.

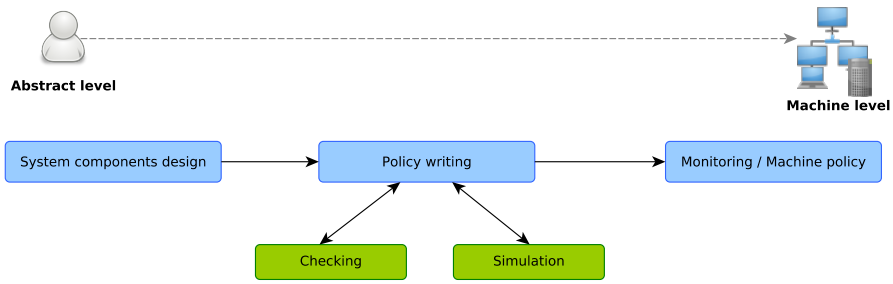


Figure 4 – AccLab workflow

Framework architecture. Figure 5 shows AccLab global architecture. The system architecture component diagram is translated automatically into `AAL` declarations, policies and obligations in natural language are translated manually into `AAL` policies. After that we perform a type analysis on the `AAL` code, then the `AAL` code is translated automatically into `FOTL` formulas. We check the consistency and the compliance of the policies using a prover (TSPASS). The simulation engine uses `AAL` code and agents behaviors in order to run the simulation (that can be connected with external tools). The export to machine level can be either to export the policies into A-PPL [Azr+15] (an extension of XACML) code and send them to the A-PPL engine, or generate a specification for `AccMon` framework (Accountability monitoring framework see Section 5.4).

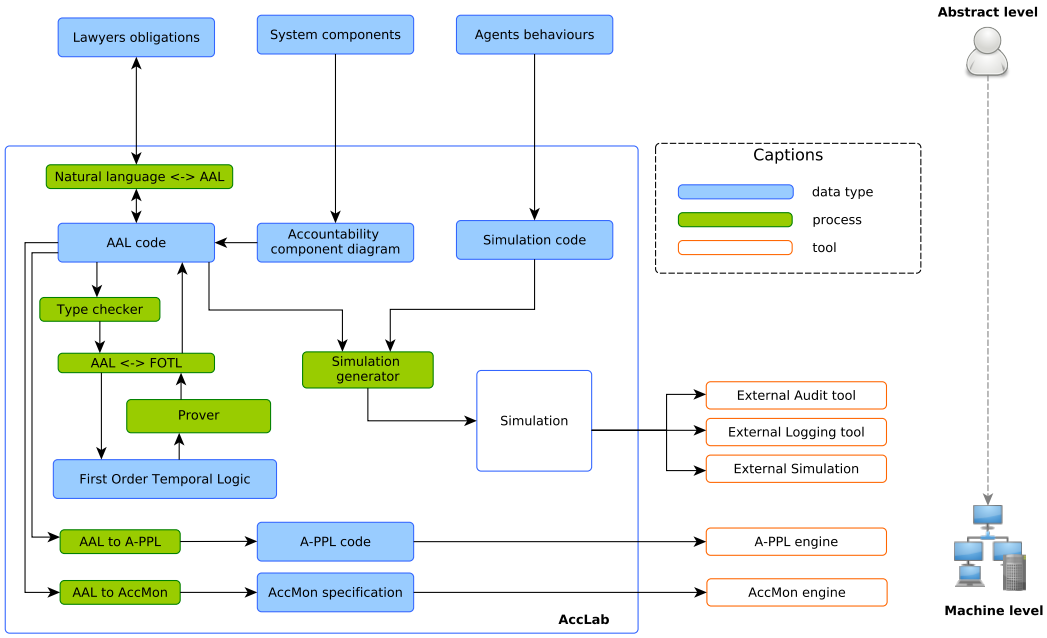


Figure 5 – AccLab architecture

5.2.2 AccLab IDE

As mentioned before the `AccLab` IDE is a web interface that provides a component diagram editor and tools to work with `AAL` language.

The Figure 6 shows the main interface of `AccLab`.

1. Project explorer tree that contains files and folders of the workspace. A version control (SVN) is integrated into `AccLab` workspace.
2. Diagram editor: the diagram editor allows to create actors with their provided and required services, connect them with each other and create `AAL` policies. It also allows to perform checks on `AAL` policies and to control agents in the simulation mode.

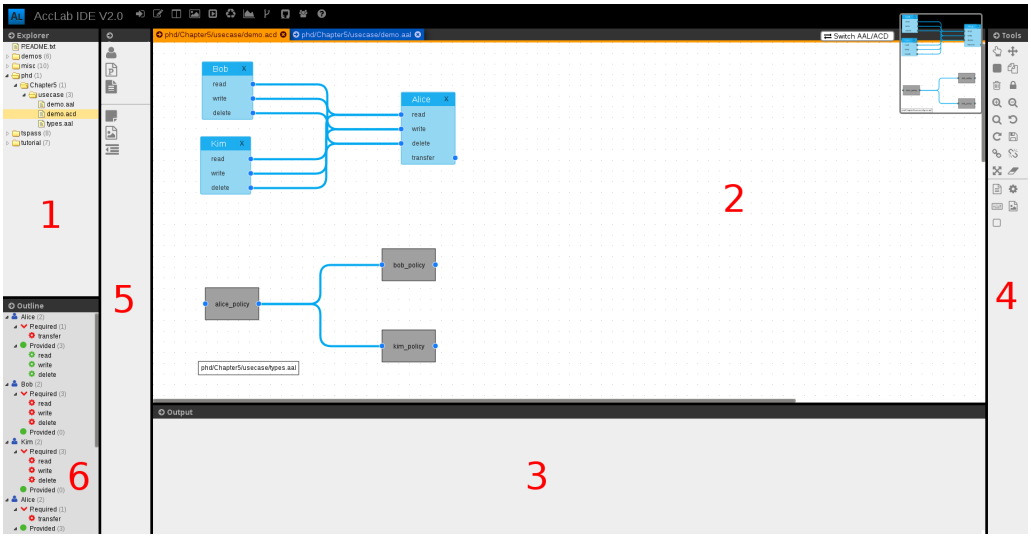


Figure 6 – AccLab UI (component diagram view)

3. Output: shows the compilation results and logs.
4. Tools: contains different tools to manipulate the component diagram/`AAL` editors.
5. Diagram components: contains the elements to add in the diagram.
6. Outline: shows the elements contained in the current component diagram.

The Figure 7 shows the `AAL` view of `AccLab`, the gray panel (1) is the `AAL` editor that provides syntax highlighting, syntactic and semantic linter and auto-completion. The output (2) contains the compilation results.

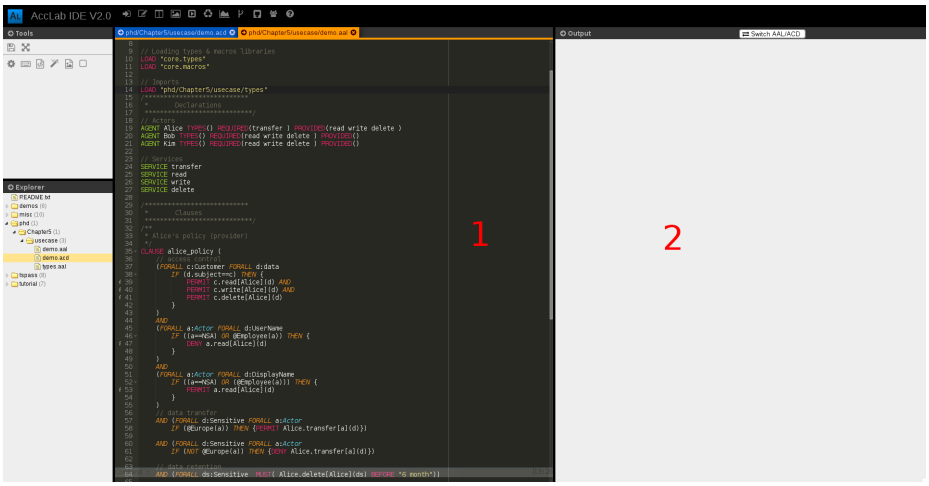


Figure 7 – AccLab UI overview

5.2.3 Simulation and monitoring

One idea behind `AccLab` is to see accountability in action, one way to achieve that is to be able to run simulations. In this part we present the simulation model of `AccLab` and the usage of a real monitoring engine.

5.2.3.1 Simulation model

Below we present the simulation model of `AccLab`. Each agent in the system is wrapped by a reference monitor which acts as a proxy and intercepts all incoming and out-coming messages of an agent, reference monitors communicates with each other via a component that simulates the network.

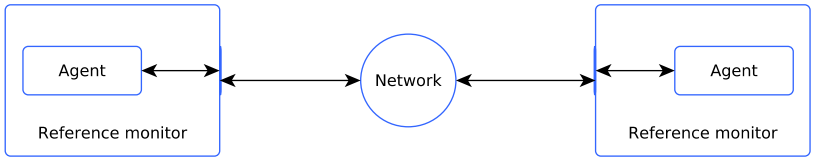


Figure 8 – AccLab simulation model

Definition 16 (Basic communication protocol)**.** The general communication schema is as follows: an actor performs a local action and produces a message to another actor.

$$ACTOR_1 \xrightarrow[Produced\ message]{local\ action} ACTOR_2$$

The detailed communication using the reference monitors is as follows: the actor A_1 want to send a message *action* to the actor A_2, The actor A_1 performs a local action that produces a message $MSG(action)$ which is intercepted by his reference monitor RM_1 that logs and checks if it is a legitimate action, and forwards the message to the reference monitor RM_2 of the actor A_2 that performs the same checks as RM_1 but on his own policy and forwards the message to A_2.

$$A_1 \xrightarrow[MSG(action)]{action} RM_1 \xrightarrow[MSG(action)]{\vdash CHK(action)} Net \rightarrow RM_2 \xrightarrow[MSG(action)]{\vdash CHK(MSG(action))} A_2$$

Reference monitor. As said before, in our model each agent is wrapped by a reference monitor. An agent which is not wrapped is considered as an intruder in the system. In the following we expose the main ideas to design our reference monitors.

In [Sen+04] authors present three key guiding principles to design their monitoring algorithm:

1. A local monitor should be fast, so that monitoring can be done online.

2. A local monitor should have little memory overhead, in particular, it should not need to store the entire history of events on a process.
3. The number of messages that need to be sent between processes for the purpose of monitoring should be minimal.

In [And74] authors define a secure reference monitor in terms of satisfying three criteria:

1. Non-Bypassability: a reference monitor is always invoked during each communication.
2. Tamperproofness: a reference monitor cannot be modified by unauthorized agents.
3. Verifiability: a reference monitor can be proved correct, with a sufficient level of trust.

In the real world, non-bypassability and tamperproofness may not be guaranteed due to technical breaches. Since we want to be close to the real world, we allow these properties to be violated in our model.

We consider two types of agents :

1. Safe agent: under this hypothesis, controls performed by the reference monitor can be relaxed.
2. Unsafe agent: with this assumption the reference monitor intensify its controls.

5.2.3.2 Attacks

Below we present some known attack principles and how we model them in our simulation model.

Man-in-the Middle attack (MITM): which is one of the famous attacks principles:

1. Between two reference monitors: The attack is performed on the network, by intercepting messages between reference monitors [1]. This attack has many derivations depending on the monitors communication protocol, messages can be simply destroyed or altered. We suppose that messages are encrypted by reference monitors. Altering messages suppose for the attacker to know the encryption / decryption keys.

$$RM_1 \xrightarrow[SIG(MSG(action))]{\vdash CHK(action)} Net \xrightarrow[MSG(action)]{Attack} RM_2$$

2. Between an agent and its reference monitor: in a realistic case this scenario is totally relevant, if we consider that the agent and its reference monitor are on the same physical machine and this one can be infected by a virus. In the other case where the agent

1. In our model this attack can be simulated by the network.

and its monitor are not on the same physical machine, it can be a network attack.

$$A_1 \xrightarrow[MSG(action)]{action} Attacker \xrightarrow[MSG(action2)]{\nvdash CHK(action2)} RM_1 \xrightarrow[MSG(action2)]{\vdash CHK(action2)} \ldots$$

In this scenario the attacker intercepts messages coming from the agent, supposing that it is a legitimate action the attacker substitutes the action by another action which is not allowed, the monitor could consider it as a violation and a rectification could be triggered.

$$\ldots \rightarrow Net \rightarrow RM_2 \xrightarrow[MSG(action)]{\vdash CHK(MSG(action))} Attacker \xrightarrow[MSG(action2)]{\nvdash CHK(MSG(action2))} A_2$$

Unsafe agent attacks.

1. Hidden channels: Considering this kind of attack we intentionally violate the Non-Bypassability criteria of [And74]. In practice the monitors don't always control all communication channels, even if it's the case attackers can exploit trusted channels to dissimulate its activities.

$$A_1 \xrightarrow[MSG(action(hiddenMsg))]{action} RM_1 \xrightarrow[MSG(action)]{\vdash CHK(action)} Net \rightarrow RM_2 \xrightarrow[MSG(action(hiddenMsg))]{\vdash CHK(MSG(action))} A_2$$

2. Controlling reference monitor: This is a less frequent attack since it requires advanced technical skills to achieve it. The idea is to alter the behavior of the reference monitor that can be done by buffer over flow attacks, reverse engineering, etc.

$$A_1 \xrightarrow[MSG(EXEC(\sigma_1 \rightarrow \varphi_1 \,?\, \sigma_2[\sigma_1/\sigma_3]))]{\vdash attack} RM_1 \xrightarrow[MSG(action)]{\vdash CHK(action)} \ldots$$

3. Log corruption: Logs can be corrupted even if we could make the assumption that logs are secure and encrypted. Logs can simply be deleted, which can be problematic to perform an efficient audit. Many other scenarios about log corruption are possible, an attacker can delete his actions from logs or adds traces in logs of actions that not have been executed, etc.

5.2.4 Use case

In this part we present a use-case that illustrates the use of AccLab. Figure 9 shows a component diagram that contains three actors (Alice, Bob and Kim) and their policies.

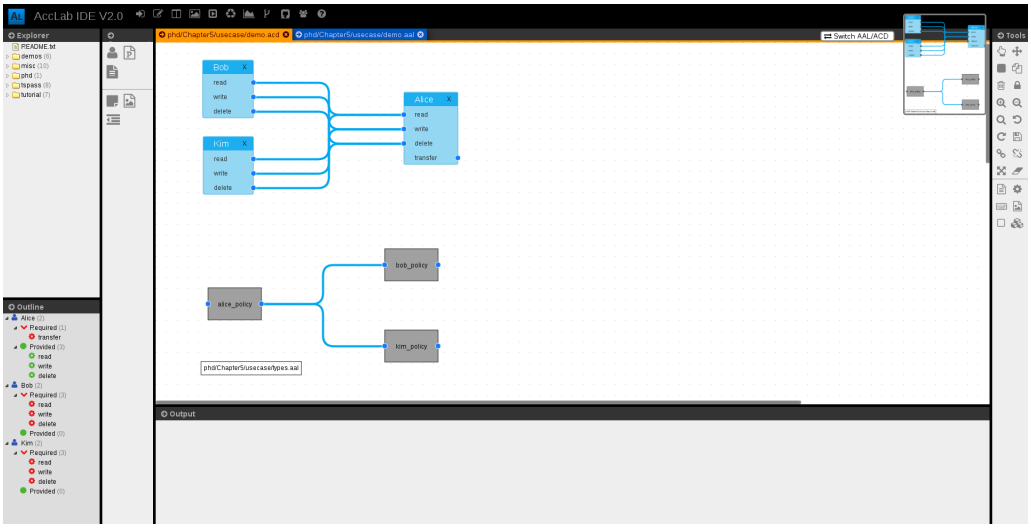


Figure 9 – Use-case: AccLab component diagram editor

Bellow the declaration part of this system in AAL and actors policies.

```
// Services
SERVICE transfer
SERVICE read
SERVICE write
SERVICE delete

// Actors
AGENT Alice TYPES() REQUIRED(transfer) PROVIDED(read write delete)
AGENT Bob   TYPES() REQUIRED(read write delete) PROVIDED()
AGENT Kim   TYPES() REQUIRED(read write delete) PROVIDED()
```

```
/**
 * Alice's policy
 **/
CLAUSE alice_policy (
    // access control
    (FORALL c:Customer FORALL d:data
        IF (d.subject==c) THEN {PERMIT c.read[Alice](d) AND
                PERMIT c.write[Alice](d) AND PERMIT c.delete[Alice](d)
    })
    AND
    (FORALL a:Actor FORALL d:UserName
        IF ((a==NSA) OR @Employee(a)) THEN {DENY a.read[Alice](d)})
    AND
    (FORALL a:Actor FORALL d:DisplayName
        IF ((a==NSA) OR (@Employee(a))) THEN {PERMIT a.read[Alice](d)})
    AND
    // data transfer
    (FORALL d:Sensitive FORALL a:Actor
```

```
18          IF (@Europe(a)) THEN {PERMIT Alice.transfer[a](d)})
19
20      AND (FORALL d:Sensitive FORALL a:Actor
21          IF (NOT @Europe(a)) THEN {DENY Alice.transfer[a](d)})
22
23      // data retention
24      AND (FORALL ds:Sensitive  MUST( Alice.delete[Alice](ds) BEFORE "6
        month"))
25
26      AUDITING MUST(dpa.audit[Alice](log) BEFORE "1 month")
27      IF_VIOLATED_THEN MUST(dpa.sanction[Alice](d))
28  )
29
30  /**
31   * Bob's policy
32   **/
33  CLAUSE bob_policy (
34      // access control
35      (FORALL e:Employee FORALL d:DisplayName DENY e.read[Alice](d)) AND
36      (FORALL e:Employee FORALL d:DisplayName PERMIT e.read[Alice](d))
37
38      // data transfer
39      AND (FORALL d:Sensitive FORALL a:Actor
40          IF (@Germany(a) OR @France(a)) THEN {PERMIT Alice.transfer[a](d
        )})
41
42      // data retention
43      AND (FORALL ds:Sensitive MUST(Alice.delete[Alice](ds) BEFORE "4
        month"))
44  )
45
46  /**
47   * Kim's policy
48   **/
49  CLAUSE kim_policy (
50      // access control
51      (FORALL e:Employee FORALL d:UserName DENY e.read[Alice](d)) AND
52      (FORALL e:Employee FORALL d:DisplayName PERMIT e.read[Alice](d))
53
54      // data transfer
55      AND (FORALL d:Sensitive FORALL a:Actor
56          IF (@US(a)) THEN {PERMIT Alice.transfer[a](d)})
57
58      // data retention
59      AND (FORALL ds:Sensitive MUST(Alice.delete[Alice](ds) BEFORE "6
        month"))
60  )
```

Checking policy's consistency. The Figure 10 shows consistency detection in Bob's policy. The error here is that we allow and deny employee to read their display name from Alice at the same time.

```
(FORALL e:Employee FORALL d:DisplayName DENY e.read[Alice](d)) AND
(FORALL e:Employee FORALL d:DisplayName PERMIT e.read[Alice](d))
3..
```

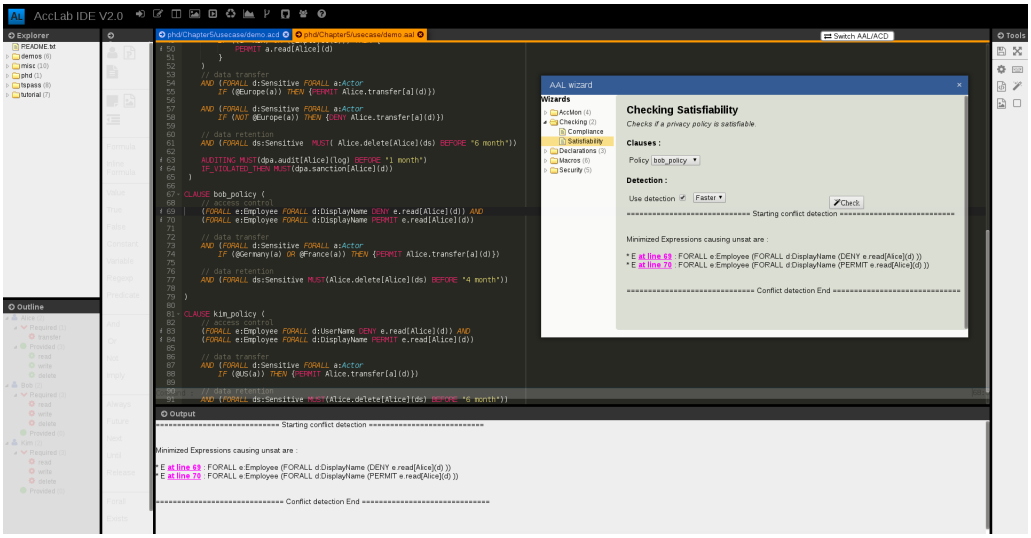


Figure 10 – AccLab consistency detection

Checking policies compliance. The Figure 11 shows compliance detection between Bob's policy and Alice's policy. Here Alice specifies 6 months for data retention period while Bob specifies 4 months.

```
// data retention
    AND (FORALL ds:Sensitive MUST(Alice.delete[Alice](ds) BEFORE "6
    month"))
...
// data retention
    AND (FORALL ds:Sensitive MUST(Alice.delete[Alice](ds) BEFORE "4
    month"))
```

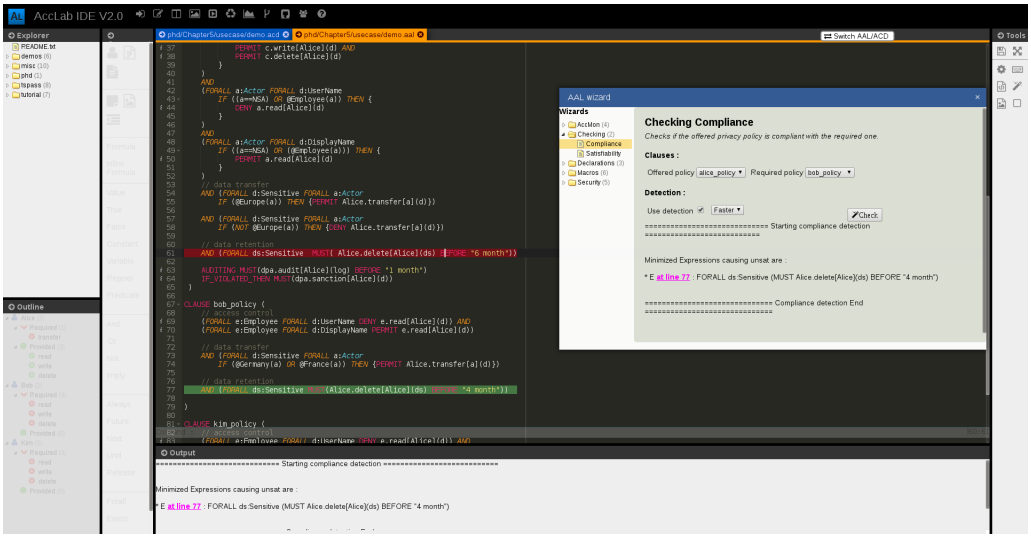


Figure 11 – AccLab compliance detection

Simulation. The Figure 12 shows simulation environment of `AccLab`, with terminals emulators for each actor.

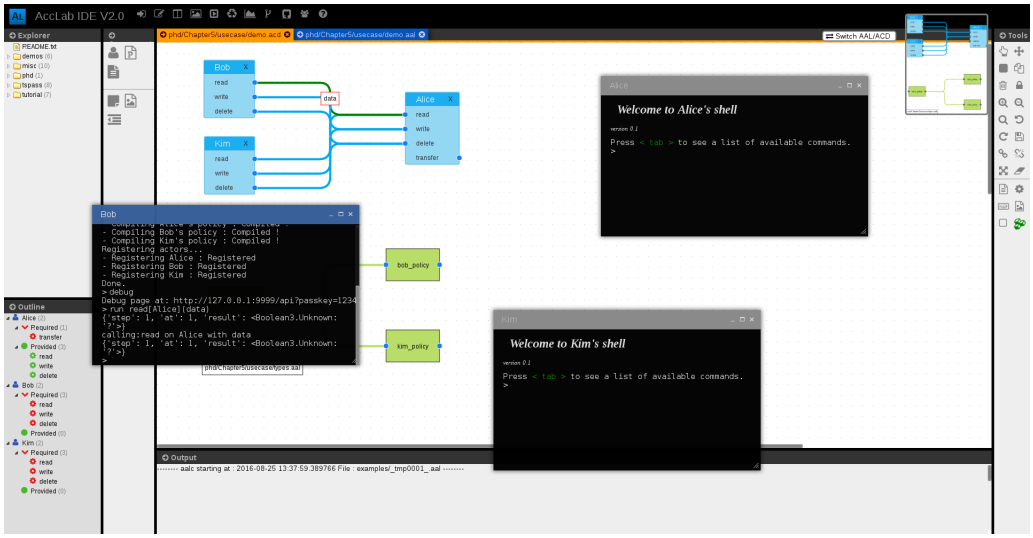


Figure 12 – AccLab simulation

5.3 Fodtlmon: a monitoring framework for *FO-DTL*^{3}

`FodtlMon` is the core monitoring framework for *FO-DTL*^{3} . The first version was released on github (`github.com/hkff/FodtlMon`) on February 2016 under GPL3 license. The last release of `FodtlMon` is version 1.2 which was released on August 2016, this version includes mainly monitor optimizations and some bug fixes. `FodtlMon` has 1600 sloc. The framework has a modular architecture, as the Figure 13 shows at the top level we have the *FO-DTL*^{3} monitor which extends `FOTL` and `DTL` monitors and both extends `LTL` monitor. Note that all monitors use a Three-valued logic. The framework includes a server that exposes a REST API to create and run monitors.

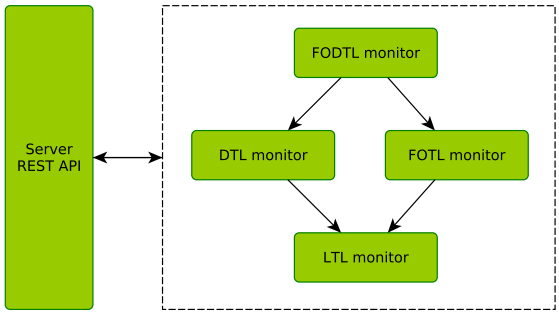


Figure 13 – Fodtlmon architecture

Below the concrete syntax of *FO-DTL*^{3} used in `FodtlMon`.

G Always, F for Future and X for Next.

ψ	::=	`true` | `false` | ψ | ψ(&|'|'|=>)ψ | φ	(`propositional formulas`)
	|	$X\psi$ | $G\psi$ | $F\psi$ | $\psi\ \mathcal{U}\ \psi$ | $\psi\ \mathcal{R}\ \psi$	(`temporal formulas (future)`)
	|	$![x_1 : type_1 \ldots x_n : type_n]\psi$	(`existential quantifier`)
	|	$?[x_1 : type_1 \ldots x_n : type_n]\psi$	(`universal quantifier`)
	|	$@name(\psi)$	(`distribution operator`)
φ	::=	$P(t^*)$ | $I(t^*)$ | $f(\varphi^*)$	(`predicates and functions`)
	|	`C` | `V`	(`constants and variables`)

Table 13 – Fodtlmon syntax

A trace is a set of events and an event is composed of a set of predicates.

```
event : {Predicate(args) | ....}
trace : {event1; event2; .... }
```

Example 13. Here a simple example using `LTL` monitor, `-f` specifies the formula, `-t` the trace and `-1` to use the `LTL` monitor.

```
python3 mon.py -f "F(w('x'))" -t "{w(a)};{w(a)};{w(x)}; {w(a)}" -1
>> Result Progression: True after 3 events.
```

Below the available monitors:
— LTL monitor: -1 or - -ltl
— FOTL monitor: -2 or - -fotl
— DTL monitor: -3 or - -dtl
— FODTL monitor: -4 or - -fodtl
— Instrumented FOTL monitor: -5 or - -ifotl

The option `-opt` allows to specify the optimization to use (0: for the simplification rules, 1: use the TSPASS solver, 2: to use Fixpoint reduction and rules, 3: to use both simplification and Fixpoint)

Interpreted predicates and functions. We can define new interpreted predicates and functions, the general form is presented below:

```
# A predicate is a class that inherit from IPredicate
class my_predicate(IPredicate):
        # We only need to redefine the eval function
    def eval(self, valuation=None, trace=None):
        # Call the eval function of the super class in order to handle
        # the arguments correctly
        args2 = super().eval(valuation=valuation, trace=trace)

        # Make your custom computation

        # Return a boolean value
        return <boolean value>

# A function has the same structure as a predicate except that we
    inherit from
```

```
# Function class
class my_function(Function):
    def eval(self, valuation=None, trace=None):
        args2 = super().eval(valuation=valuation, trace=trace)
        # Return a value of any type
        return <value>
```

Example 14. We define the addition function and a predicate that checks if a number is greater than zero.

```
# An interpreted predicate that checks if the given argument is greater
     than 0
class Gtz(IPredicate):
    def eval(self, valuation=None, trace=None):
        args2 = super().eval(valuation=valuation, trace=trace)
        if len(args2) > 0:
            return int(args2[0].name) > 0
        else:
            raise Exception("Missing arguments")

# A function has the same structure as a predicate except
class Add(Function):
    def eval(self, valuation=None, trace=None):
        args2 = super().eval(valuation=valuation, trace=trace)
        if len(args2) > 1:
            return int(args2[0].name) + int(args2[1].name)
        else:
            raise Exception("Missing arguments")
```

Then we can use them in a formula:

```
python3 mon.py -f "Gtz(Add('1', '2'))" -t "{}" -2
>> Result Progression: True after 1 events.
```

For more information please refer to the framework homepage.

5.4 Monitoring accountability with AccMon

The goal of `AccMon` is to provide means to monitor accountability policies in the context of a real system. The first version was released on github (`github.com/hkff/AccMon`) on February 2016 under GPL3 license. The last release of `AccMon` is version 1.1 which was released on April 2016. This framework is based on Django which is an open-source web application framework written in Python and it is based on the model-view-controller (MVC) pattern. `AccMon` has 1800 sloc. `AccMon` allows to specify policies that are applicable to network traffic, web application code and external components via plugins.

Global architecture. As the Figure 14 shows, `AccMon` acts as a middleware in the Django framework, it intercepts and logs client's HTTP requests, server's requests processing and responses. On the web application side the developer can configure the framework to intercept function/method calls and databases access. `AccMon` can act

as a daemon and can be interconnected with external tools, the property to monitor on the tool is defined in `AccMon` and the tool sends log events to `AccMon` via HTTP calls. The framework can also be connected with external hardware such as `Arduino` electronic boards (an Arduino plugin was developed and is available in `AccMon` plugins).

Arduino is an open-source prototyping platform based on easy -to-use hardware and software.

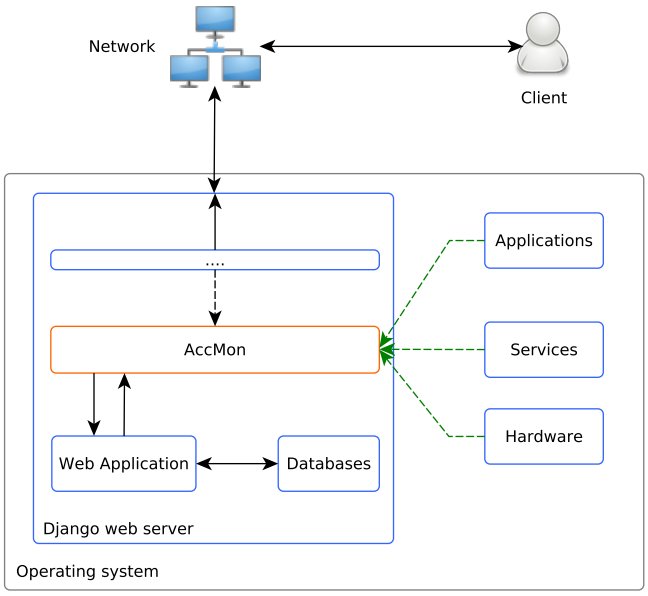


Figure 14 – AccMon global architecture

Framework architecture The Figure 15 shows AccMon internal architecture which is composed with:

1. *Monitors:* a monitor is defined by an id, an Fodtl formula and control type (posteriori or realtime).

```
# Add a rule (monitor) in the system
Sysmon.add_<http|view|response>_rule(<name>, <Fodtl_formula>,
    description="",
        control_type=Monitor.MonControlType. <REAL_TIME|
    POSTERIORI>)
```

2. *Logging:* a log attribute is defined by a name, a description and an evaluation function that has five arguments (request, view, args, kwargs, response) and which will be called automatically by the framework if the logging attribute is enabled.

```
# Logging attribute evaluation function
def fx(request, view, args, kwargs, response):
        # Do some computations
        # ...
        # Must return an Fodtl Predicate
        return P("<name>", args=[Constant*]))

# Creating a logging attribute
```

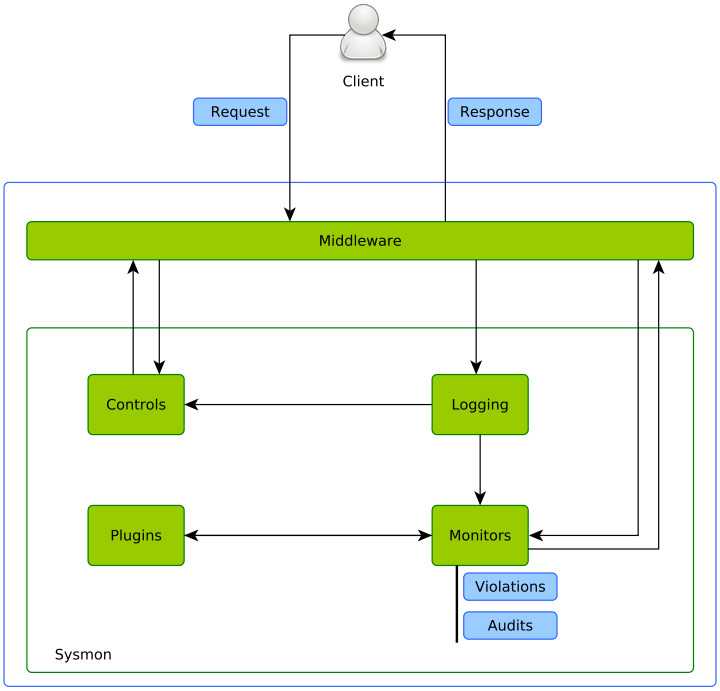


Figure 15 – AccMon architecture

```
lg = LogAttribute(<name>, description="...", enabled=<True|False
    >, eval_fx= fx)

# Adding the logging attribute to the system
Sysmon.add_log_attribute(lg, target=Monitor.MonType.<HTTP|VIEW|
    RESPONSE>)
```

3. *Controls:* defines security checks that can be extended by the developer. Below an example of an input sanitizer in order to prevent some XSS attacks:

```
class XSS(Control):
        def prepare(self, request, view, args, kwargs):
                data = getattr(request, request.method)
        for key in data:
                mutable = data._mutable
            data._mutable = True
            data[key] = sanitize(data.get(key))
            data._mutable = mutable
```

4. *Plugins:* connect `AccMon` with external components which allows to define monitors that monitor formulas for external software/ hardware in `AccMon`. The framework expose an interface in order to receive events from external components.

Usage First we need to define what we want to log, `AccMon` comes with predefined logging attributes that can be enabled and disabled at runtime. The developer can also define custom attributes to log. Here a basic example :

```
# The evaluation function of the log attribute
# Here we return a predicate that looks like : USER('user_name')
def username(request, view, args, kwargs, response):
        return P("USER", args=[Constant(request.user)])

# Create the log attribute
username_log = LogAttribute("UserName", enabled=True, eval_fx=username)

# Register the attribute in the monitoring system and make it available
# on HTTP request level.
Sysmon.add_log_attribute(username_log, target=Monitor.MonType.HTTP)
```

Next the developer can define custom interpreted predicates and functions that will be used in the monitoring formulas (this feature is provided by FodtlMon). Here an example that checks if the corresponding user (django user) of a given id has a certain username.

```
class UserEq(IPredicate):
    # Compare the username of a user for a given id with a given
    username.
    # Usage in a formula : UserEq(usser_id, user_name)
    def eval(self, valuation=None):
        args2 = super().eval(valuation=valuation)
        u1 = User.objects.filter(id=args2[0].name).first()
        return u1.usernmae == args2[1].name
```

Finally, the developer defines monitoring rules using Fodtl formulas. Here some examples :

— Add a rule on HTTP request:

```
# Add a rule on HTTP request
Sysmon.add_http_rule("UserProfile", control_type=Monitor.
    MonControlType.REAL_TIME,
"G( ![id:UIDL uname:USER req:GET]( ReqIn(r\"taskManager/profile
    /\", req) =>
 UserEq(id, uname)) )" )
```

— System interfacing example: In this example each time the user uses the change directory 'cd' command of the operating system, the system sends the event with the path to `AccMon`.

```
# Check if the path is not in /root/*
remote.Remote.add_rule("cdroot", "G( ![path:cd]( ~Regex(path, r
    \"/root/*\")) )")
```

— Hardware interfacing example (Arduino) : In this example we built a laser detector using an Arduino board, a laser diode and a light sensor. Basically the laser diode points into the light sensor which sends to the arduino board a value of the light intensity. The code on the arduino board (written in C language) simply reads values from the light sensor and print them in the serial port, in the form of a predicate `LIGHT(value)`.

The `AccMon` arduino plugin read the events from the serial port and append them to arduino monitors traces.

```
# The light intensity should always be greater than 10
arduino.Arduino.add_rule("light", "G( ![x:LIGHT]( Gt(x, '10') ))"
    )
```

Screenshots. The Figure 16 presents an overview of the running monitors with their name, location, status and violations.

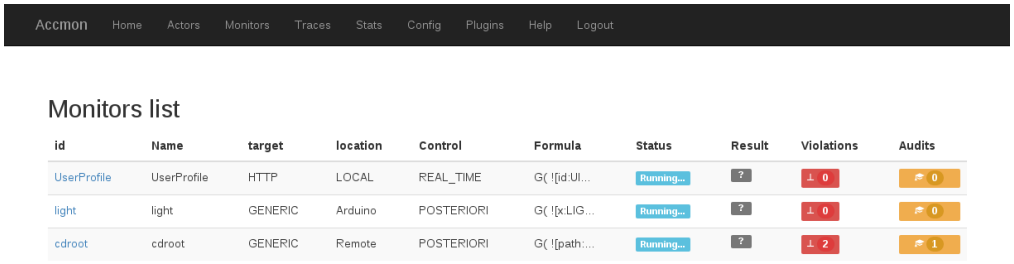


Figure 16 – AccMon monitors overview

The Figure 17 shows a view of the traces collected at runtime, each row contains the timestamp and a list of events.

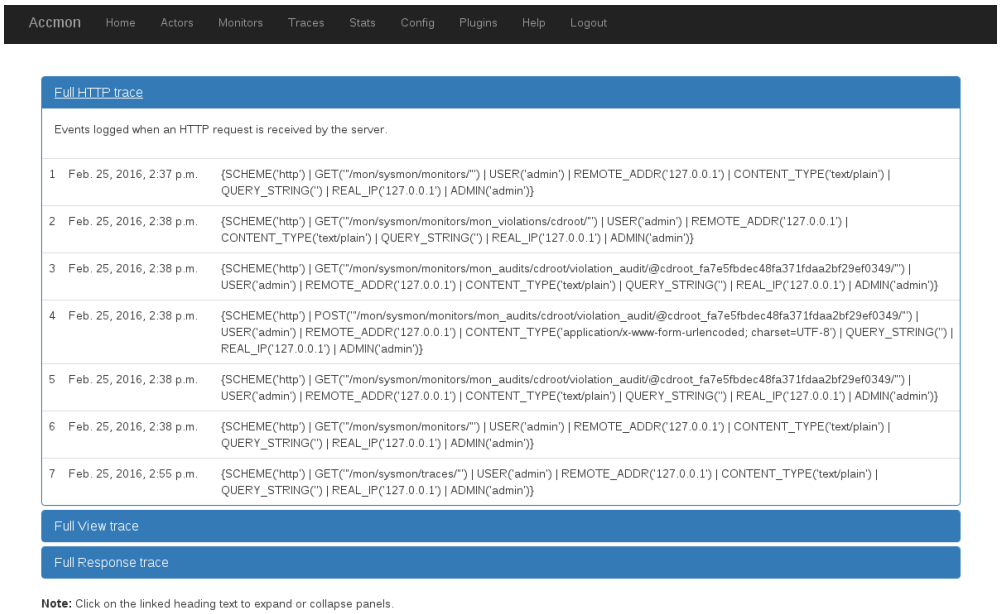


Figure 17 – AccMon traces

5.5 PyMon: a python monitoring framework

We applied our monitoring framework to Python code. `PyMon` allows to monitor function and method calls. The prototype of this framework can be found on github (`github.com/hkff/pymon`) and it is released under GPL3 licence. `PyMon` has 400 sloc.

Function/method calls. For each function/method decorated with *mon_fx* decorator, we attach a monitor that will be called on function calls.

Here for example we want to ensure that, if one of the arguments is a string then the function should return a string.

```
@mon_fx("G( (?[x:ARG] str(x)) => (![y:RET] str(y)) )")
def concat(a, b):
  # Do obscure stuffs ...
  return result
```

A type system for Python. Python is a dynamic type language, with static type annotations on source code level support, but the annotations have no effects on runtime. Here a simple example of an add function defined on integers:

```
# 1. Without type annotations
def add(a, b):
    return a + b

>> add(1, "2")
>> "TypeError: unsupported operand type(s) for +: 'int' and 'str'"

# 2. With type annotations
def add(a: int, b: int):
    return a + b

>> add(1, "2")
>> "TypeError: unsupported operand type(s) for +: 'int' and 'str'"
```

Calling the function with an *integer* and a *string* leads to a runtime type error in both cases.

Using `PyMon`:

```
@SIG("(a:int, b:int) -> int")
def add(a, b):
        return a + b

>> add(1, "2")
>> "Arguments type Error !  Expected <(int('a')) & (int('b'))>"
>>                          "Found : [a: <int>, b: <str>]"
```

Currently we support all basic python type, classes and lists.

Distributed monitoring. As `PyMon` relies on `FodtlMon` engine, it supports distributed monitoring. In the following example, the monitor of the function foo checks that the argument a of the function bar is always an integer.

```
@mon_fx(["G(int('a') & @bar(int('a')))"], name="foo")
def foo(a, b):
    return a

@mon_fx("X(X(int('a')))", name="bar")
def bar(a, b):
    return a
```

5.6 Conclusion

In this chapter we presented a set of tools that handle accountability at different levels of software development cycle. The goal is to have an end to end framework that helps developers to integrate accountability from the design time to the implementation.

Many improvements are possible, first we want to have a better synergy between the different tools, especially the link between `AccLab` and `AccMon`. The gap between the abstract architecture and the implementation is huge, which makes an automated mapping very difficult. However since `AAL` is an abstract language, we can easily describe technical behaviors close to the implementation, but we lose the link with the abstract level. One approach can be refinement technique like Event-B method. Refinement consists to represent systems at different abstraction levels and to use proof to verify consistency between refinement levels.

Concerning `AccMon`, one major improvement is the synchronization frequency between actors. Currently, the knowledge vectors are synchronized only when actors communicate with each other. Another point is about the monitoring engine `FodtlMon`, as we have seen in Chapter 4 the monitoring technique is not complete and needs to be coupled with a theorem prover. Currently, `FodtlMon` supports this approach by using the *TSpass* prover, but calling a prover at each monitoring step is not efficient and no heuristic was developed at this time.

Part III

ACCOUNTABILITY: BEYOND ACCOUNTABILITY

This part concludes this manuscript but doesn't conclude this thesis. Protecting our privacy is more than a thesis work, a computer science topic or a technical problem. It is a way of thinking, a way of living, and it is about ethics and human morality.

" ."

6

CONCLUSION AND FUTURE WORK

"To know, is to know that you know nothing. That is the meaning of true knowledge."

— **Socrates**

During this thesis I had the chance to work on interesting questions and bring some solutions, but these solutions are partial and bring many other questions.

We started with a simple question: how do we ensure that our privacy is guaranteed in the digital world? We answered by checking that the privacy policies of the actors that handle our data are compliant with the law and regulations. But from this answer two more issues appear: (1) first regulations and the privacy policies that actors expose publicly are written in natural language which makes the comparison more complicated due to the ambiguity and the multiple interpretation of natural language; (2) the second issue is that we have no guarantees that the privacy policies are implemented correctly because their implementation is written in machine language. From the second issue another point is raised which is on the perfect security, we agree that no system is perfectly secure so what happens if a violation occurs? And who is the responsible regarding the law? This is where the accountability enters in action, in the sense that we have to define who is the responsible in case of violations and what we should we do. At this point we answered the sub-issues but the two main issues are still without an answer. To deal with the problem of natural language ambiguity and the gap between natural language and machine language, we designed a language close to natural language and formal enough to be interpreted by a machine, and we included the accountability in order to handle the previous sub-issues. We called this language `AAL` (Abstract accountability language). To return to the second main problem of how we guarantee that policies are correctly implemented, we decided to use monitoring techniques. Since `AAL` semantics is based on a first order temporal logic, we need to have techniques to monitor such logic. Thus we introduced *FO-DTL*^{3} which is a first order distributed temporal logic and we used progression technique to monitor this logic. We raised the problem of completeness of this technique and we came up with another monitoring technique called co-inductive monitoring.

The Figure 18 illustrates the flow of reasoning presented previously, the red boxes correspond to the questions/issues, the green boxes correspond to the solutions, the yellow boxes correspond to the potential

As you can see, there are more red boxes than green ones . . .

solutions or future work and the blue boxes are the tools that we developed.

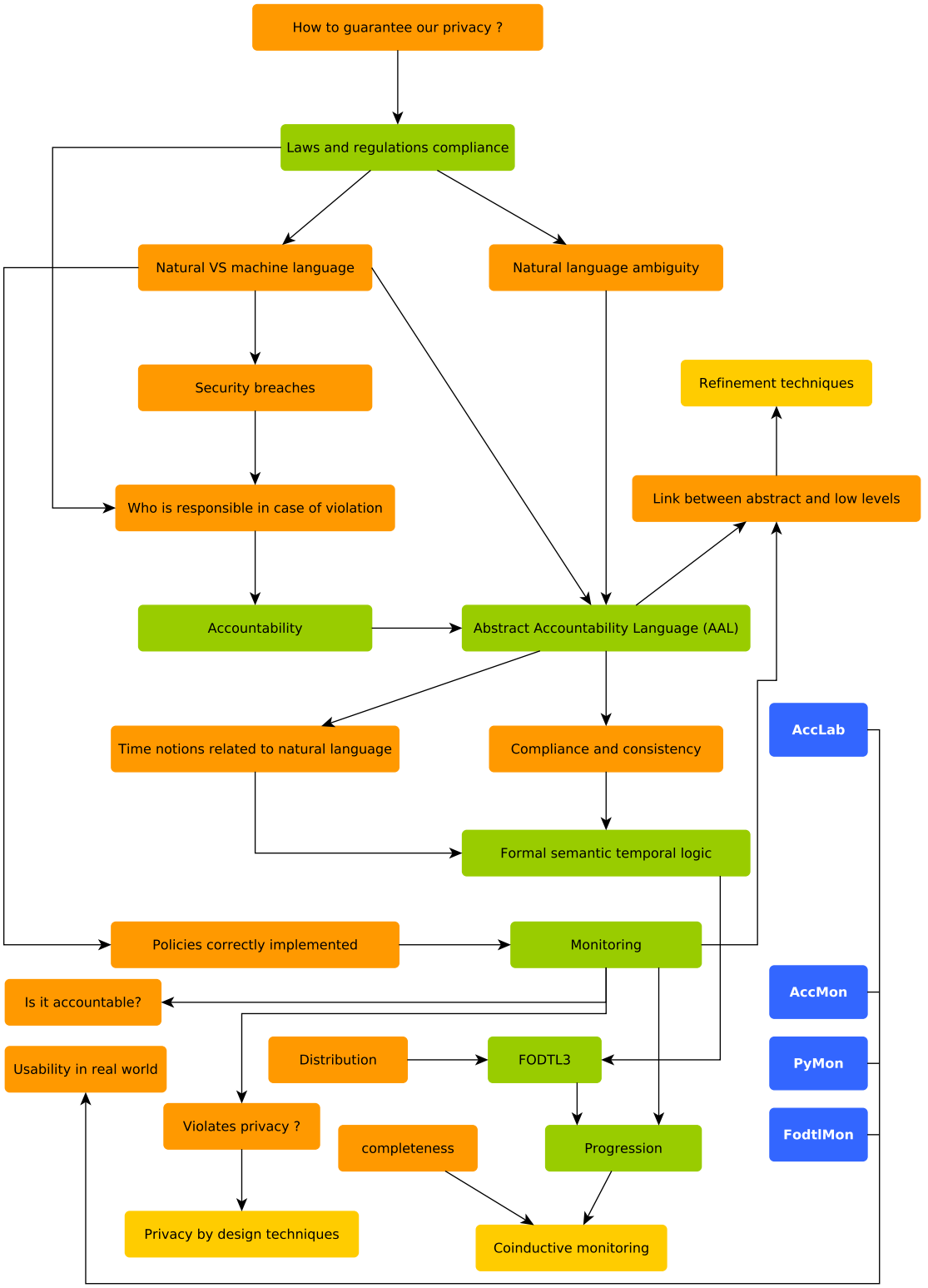


Figure 18 – Thesis flow of reasoning

Beyond accountability. Accountability in the digital world implies many things such as privacy and security and the problem is beyond technical challenges, it's educational, political, sociological and at some point a philosophical challenge. Accountability is not limited to privacy, and can be applied to larger domains, accountability policies can describe not only privacy policies but also more general protocols.

Can an accountability system be accountable itself? At some point we have to trust someone or something but the question is at which point shall we do that? The Figure 19 illustrates perfectly this issue.

1. To control the police, there is the police of police
2. And to control the police of police ?
3. A police of the police of police
4. And a police of the police of the police of the police of the....

Image reproduced by kind permission of Mr Xavier Gorce

Figure 19 – Police of police

Currently, in order to be accountable we need to be monitored. Of course we can limit the collected data for the monitoring purpose, but conceptually the need to monitor in order to protect our privacy violates our privacy. We need to care about ethics when we develop monitoring techniques, because even if our intentions are good, theses techniques can be used for bad purposes such as surveillance and in this case it will be the opposite of the original purpose. To limit the use of monitors some work on privacy by design exists [BCM14]; [Che16] but it cannot be applicable to all situations. One approach is to develop monitoring technique that respect our privacy using privacy by design approach.

During this thesis we developed a set of tools in order to answer to accountability issues. `AccLab` can be used to validate accountability policies and to check if a system is accountable. It can be used to check if a system policy is compliant with regulations like the GDPR. However, the usability for non specialists represent an issue for the adoption of the tool in the real world. `AccMon` and `PyMon` allow to monitor real world applications, but we still need to improve their

performance in order to be scalable. Even if these issues are fixed, we still need to find a way to convince societies and developers to use these tools. Imposing the use of such tools is not realistic.

Future work. As we can see in the Figure 18 there are more questions than answers, there is still a lot of work. In addition to the previously cited ameliorations that concern the `AAL` language usability and semantics, monitoring technique and tool support, there are also many paths to explore.

Some characteristics are specifics of human behaviors (for instance, moral dilemma or force majeure) and cannot be modeled or are out of the scope of automation. Many works consider that true accountability cannot be completely automatic and humans (an auditor, a judge, etc) should be involved in the process. According to [Sch99]: "Given that the notion of accountability is not built on the illusion that power is subject to full control ...". A fully automated solution would be equivalent to a preventive security solution, all violation cases and countermeasures are known and decided in advance. In our approach human behavior is defined by actions connected to a virtual agent (as in [Mét09]) under the control of the real human, thus it will be transparent here. These actions may occur mainly in the audit and rectification expressions. With the emergence of deep learning and artificial intelligence techniques, maybe we can start to consider automatic remediation where human actions and moral dilemma are performed by artificial intelligence .

As we saw previously, monitors that are based on progression technique are not complete and we need to combine it with a theorem prover. However this introduces an important overhead which is not desirable when we deal with online monitors. Thus we are working on a new monitoring technique that uses inference rules based on the standard semantics of `LTL`. The idea is to combine the satisfiability (like in the progression technique) and the validity (like in the theorem prover). The idea is to deduce a monitoring algorithm combining two inference systems, one for validity, another one for satisfaction. A similar approach exists in [SRA03], where authors use coinduction-based technique to generate an optimal monitor that can detect good and bad prefixes incrementally for a given trace.

Nowadays we are witnessing the democratization of Internet of Things (IOT). These objects collect data from the digital world but also from the physical world, and most of them offer the possibility of control and remote access. As a result, more and more personal data are sent to the cloud, which is not without consequences. For example, disclosing information from a connected thermometer does not seem critical, it may nevertheless make possible to know whether the house is occupied or not. Even worse a mismanaged remote access to a connected thermostat or a connected hot plate can represent a real physical danger. We

plan to explore monitoring possibilities using `AccMon` that provides distributed monitoring capabilities which are suitable for IOT. We already developed an arduino plugin for `AccMon`.

Perhaps the answer to privacy issues today is to disconnect your computer from the Internet, shut it down, burn your hard drive and throw it far away in the river... and go to live in a cave! But I hope that research will bring other answers tomorrow.

Part IV

APPENDIX

A

GETTING STARTED WITH ACCLAB

a.1 Installing AccLab

To use the ltl prover, you need to put the following executable files (tspass, fotl-translate) in `tools /your_platform/ (linux/mac/win)` `TSpass` binaries are provided for linux x64 and mac x64 in the folder `tools/<platformeName>/`. For other platformes you have to compile tspass source code. The last version of `TSpass` can be found in [Mic]. The source code for `TSpass` version 0.95-0.17 is provided with this tool. AAL Syntax highlighting modes for emacs, intellij, nano and ace, can be found in `tools/utils/`. If you want to run aalc using a symbolic link you need to set the environment variable `ACCLAB_PATH` : `export ACCLAB_PATH=<AccLab_install_dir>`. You need python3.4.0 or greater.

a.1.1 Using AAL compiler "aalc"

In this part we see how to use `aalc` which is the command line backend of `AccLab`.

Listing 7 – aalc options

```
root@root/:$ python aalc.py
AAL tools set V 2.1 . aalc is a part of Acclab tool.
For more information see AccLab home page
 Usage : aalc.py [OPTIONS]
  -h    --help                  display this help and exit
  -i    --input= [file]         the input file
  -o    --output= [path]        the output file
  -c    --compile               compile the file, that can be loaded after
                                 using -l
  -m    --monodic               apply monodic check on aal file
  -s    --shell                 run a shell after handling aal program
  -k    --check                 perform a verbose check
  -l    --load                  load a compiled aal file (.aalc) and run a
                                 shell
  -t    --fotl                  translate the aal program into FOTL
  -r    --reparse               reparse tspass file
  -r    --recompile             recompile the external files
  -b    --no-colors             disable colors in output
  -x    --compile-stdlib        compile the standard library
  -d    --hotswap               enable hotswaping (for development only)
  -a    --ast                   show ast tree
  -u    --gui= [port]           run the gui on the specified port
  -n    --no-browser            do not start the web browser
  -q    --timeout= [n]          TSPASS prover timeout (in seconds)
```

Bellow we describe the different tool options.

— -i –input= [file]: this argument allows to specify the input file to be processed, the program accepts two kind of files (`*.aal`) that

correspond to AAL programs and (*.tspass) that contains FOTL formulae using TSpass syntax.
— -o –output= [path]: this argument allows to specify the stdout output file.
— -c –compile: this option works only with AAL files. If this argument is specified, the compiler will compile the AAL program into compiled AAL file format (*.aalc). As said before, in an AAL program we can load external AAL files. In order to optimize the interpretation of AAL programs, when the compiler find a reference to an external AAL file (LOAD "aalfile), it tries to lookup for the compiled version of the AAL file first and load it if it exists, otherwise it will interpret the external AAL file. This is particularly useful for AAL libraries that do not need to be reinterpreted each time, thus they are compiled once and the compiler uses the compiled version which decrease considerably the interpretation time.
— -r –recompile: if this argument is specified, the compiler will recompile the external compiled AAL files if any before using them.
— -s –shell: run a shell after handling aal program
— -l –load load a compiled aal file (.aalc) and run a shell
— -m –monodic: apply monodic check on aal file
— -k –check: perform a verbose check
— -t –fotl translate the aal program into FOTL
— -r –reparse reparse tspass file
— -b –no-colors disable colors in output
— -x –compile-stdlib compile the standard library
— -d –hotswap enable hotswaping (for development only)
— -a –ast show ast tree
— -u –gui= [port] run the gui on the specified port
— -n –no-browser do not start the web browser
— -q –timeout= [n] TSPASS prover timeout (in seconds)

a.1.2 Writing your first AAL program

Let consider the following scenario, we have three actors :
— cloud storage service: let call it css which is a cloud service provider.
— alice and bob: an end users that uses css service.

The css offers the following services: read (a user reads some data form css server), store (a user stores some data into css server), delete (a user deletes some data from css server). css allows users to read/store/delete only their data on his server, and don't allow them to read other customers data. css can also read and delete any data from his server.

Alice want to check if `css` policy respect her privacy. Typically she want to know if she is allowed to performs some actions and if bob can read here data.

1. **Declaring actors**: first we need to declare our actors.

```
// Agents declaration
AGENT alice
AGENT bob
AGENT css
```

2. **Declaring services**: next we declare the services.

```
SERVICE read
SERVICE store
SERVICE delete
```

3. **Linking services and actors**: once services are defined, we can complete actors declarations.

```
AGENT alice TYPES(Actor) REQUIRED(read store delete) PROVIDED()
AGENT bob   TYPES(Actor) REQUIRED(read store delete) PROVIDED()
AGENT css   TYPES(Actor) REQUIRED() PROVIDED(read store delete)
```

4. **Defining policies**: next we can write our policies.

```
/*
 * Cloud storage service provider policy
 */
CLAUSE css_policy (
    FORALL d:data FORALL a:Actor

    // Allow users to read their data
    IF (d.subject == a) THEN {
        PERMIT a.read[css](d)
    } AND

    // Deny access to read other
    IF (d.subject != a) THEN {
        DENY a.read[css](d)
    } AND

    // Allow css to read/delete stored data
    PERMIT css.read[css](d) AND
    PERMIT css.delete[css](d)
)

/*
 * Alice's preferences
 */
CLAUSE alice_pref (
    FORALL d:data
    // Alice want to be able to read all her data stored on css
    IF (d.subject == alice) THEN {
        PERMIT alice.read[css](d)
        AND
        // Bob cannot read Alice's data
        DENY bob.read[css](d)
    }
)
```

5. **Writing checks**: Now we want to check if `Alice`'s privacy preferences are respected by the `css` policy. To do this, we can call the macro `validate` and passing the the clauses names as arguments. Important: Note that the order of arguments is important.

```
CALL validate("css_policy" "alice_pref")
```

a.1.3 Running the program in command line

— In order to run an `AAL` program, you have to run aalc.py.

```
root@root/:$ python aalc.py -i examples/tuto0.aal

------------------------- Monodic check -------------------------
Monodic check passed !
------------------------- Starting Validity check
     -------------------------
c1 : css_policy
c2 : alice_pref
----- Checking c1 & c2 consistency :
  -> Satisfiable
----- Checking c1 => c2 :
  -> Satisfiable
----- Checking ~(c1 => c2) :
  -> Unsatisfiable

[VALIDITY] Formula is valid !
------------------------- Validity check End -------------------------

File : examples/tuto0.aal

Execution time : 0.24277639389038086
```

— You can perform a detailed check by using (-k) argument.

```
root@root/:$ python aalc.py -i examples/tuto0.aal -k

------------------------- Start Checking -------------------------

** DECLARATIONS
[DECLARED AGENTS]   : 3
[DECLARED SERVICES] : 6
[DECLARED DATA]     : 0
[DECLARED TYPES]    : 11

*** Forwards references check
[AGENTS]   : 0
[SERVICES] : 0
[DATA]     : 0
[TYPES]    : 0


*** Unused declarations

[WARNING] Unused agent declaration : bob  ->  at line 18
[WARNING] Unused service declaration : read  ->  at line 12
[WARNING] Unused service declaration : store  ->  at line 13
[WARNING] Unused service declaration : delete  ->  at line 14
[WARNING] Unused service declaration : write  ->  at line 20
[WARNING] Unused service declaration : update  ->  at line 22
[WARNING] Unused service declaration : audit  ->  at line 23


** LOADED libraries
```

```
[LIBS] : 2

** CLAUSES
[CLAUSES] : 2

*** Miscellaneous
[PERMISSIONS]   : 3
[PROHIBITIONS   : 2

*** Sat test

---------- css_policy ----------
------------------------- Monodic check -------------------------
Monodic check passed !
------------------------ Starting  check --------------------
----- Checking c1 :
  -> Satisfiable
------------------------  check End ------------------------


---------- alice_pref ----------
------------------------- Monodic check -------------------------
Monodic check passed !
------------------------ Starting  check --------------------
----- Checking c1 :
  -> Satisfiable
------------------------  check End ------------------------
```

— To perform a monodic test on all clauses, use (-m) argument:

```
root@root/:$ python aalc -i examples/tuto0.aal -m

------------------------- Start Checking -------------------------
|css_policy | Formula is monodic ! |
|alice_pref | Formula is monodic ! |
-------------------------- Checking End ------------------------
```

— Translate AAL program into FOTL (in `TSpass` syntax) using (-t) argument:

```
root@root/:$ python aalc -i examples/tuto0.aal -t

------------------------- FOTL Translation start
     ------------------------

%%%%%%%%% START EVN %%%%%%%%%%
(
(always ![a] (Actor(a) => EQUAL(a, a))) &
(always ![a, b] ((Actor(a) & Actor(b) & EQUAL(a, b)) => EQUAL(b, a)))

%%% Types knowledge
&
always (
 ( ?[a] data(a)  ) &
 ( ?[a] actor(a)  ) &
 ( ?[a] Actor(a)  ) &
 ( ?[a] Data(a)  ) &
 ( ?[a] DataSubject(a) & (![x] ( (DataSubject(x) => Actor(x) ) )) ) &
 ( ?[a] DataController(a) & (![x] ( (DataController(x) => Actor(x) ) ))
    ) &
 ( ?[a] DataProcessor(a) & (![x] ( (DataProcessor(x) => Actor(x) ) )) )
    &
 ( ?[a] DwDataController(a) & (![x] ( (DwDataController(x) => Actor(x) )
     )) ) &
 ( ?[a] Auditor(a) & (![x] ( (Auditor(x) => Actor(x) ) )) ) &
 ( ?[a] CloudProvider(a) & (![x] ( (CloudProvider(x) => Actor(x) ) )) )
    &
```

```
 ( ?[a] CloudCustomer(a) & (![x] ( (CloudCustomer(x) => Actor(x) ) )) )
    &
 ( ?[a] EndUser(a) & (![x] ( (EndUser(x) => Actor(x) ) )) ) &
 ( ?[a] User(a) & (![x] ( (User(x) => Actor(x) ) )) )
)

%%% Action authorizations
&
always (
( ![x, y, z] (read(x, y, z) => Pread(x, y, z)) ) &
( ![x, y, z] (store(x, y, z) => Pstore(x, y, z)) ) &
( ![x, y, z] (delete(x, y, z) => Pdelete(x, y, z)) )
)

%%% Actors knowledge
&
always (
 ( Actor(alice)  ) &
 ( Actor(bob)  ) &
 ( Actor(css)  )
)

%%% Time knowledge


%%% Data knowledge
&
always (
 ( ?[d](subject(d, alice)) ) &
 ( ?[d](subject(d, bob)) ) &
 ( ?[d](subject(d, css)) )
)
)
%%%%%%%%% END EVN %%%%%%%%%%

%% Clause : css_policy
((![d] ( data(d) => ((![a] ( Actor(a) => (((( ((subject(d, a)) => (Pread
    (a, css, d))) &  ((~subject(d, a)) => (~Pread(a, css, d)))) & Pread
    (css, css, d)) & Pdelete(css, css, d))) ))) )) )
%% Clause : alice_policy
((![d] ( data(d) => ( ((subject(d, alice)) => ((Pread(alice, css, d) & ~
    Pread(bob, css, d))))) )) )

-------------------------- FOTL Translation end
    --------------------------
```

a.1.4 Advanced checks

```
/*
 * Alice's preferences
 */
CLAUSE alice_pref (
    FORALL d:data
    // Alice want to be able to read all her data stored on css
    IF (d.subject == alice) THEN {
        PERMIT alice.read[css](d)
        AND
        // Bob cannot read Alice's data
        DENY bob.read[css](d)
    }
```

```
)
```

The previous call to validate macro will gives: Satisfiable. Why? Because the predicate subject is not exclusive: subject of d can be alice and bob at the same time.

A simple way to fix it is to add the condition that the subject of d is not bob :

```
IF (d.subject == alice AND d.subject != bob) THEN {
....
```

Or we can to add the following condition manually to our check:

```
(![f] (subject(f, alice) => ~subject(f, bob))) &
```

The construction CHECK allows you to write directly FOTL formula mixed with somme AAL constructions.

- @verbose (print the generated formula).
- @buildenv (build the environment, which is a set of preconditions generated from the AAL program).
- clause(c): get the fotl translation of clause "c".
- clause(c).ue: get the fotl translation of usage part of the clause "c".
- clause(c).ae: get the fotl translation of audit part of the clause "c".
- clause(c).re: get the fotl translation of rectification part of the clause "c".
- APPLY chk(): call the check "chk".

```
CHECK c1() (
"""
% Comments in Checks starts with '%'

% Enable verbose mode
@verbose
~(
    % Build the environment
    @buildenv

    % Add extra condition
    (![f] ((subject(f, alice) )=> ~subject(f, bob))) &

    % The check ~ P => U
    (clause(css_policy))
    =>
    (clause(alice_pref))
)
"""
)
APPLY c1()
```

The result is Unsatisfiable so the formula is valide.

a.1.5 Using the shell

The shell is a useful tool for developing:

— Run the shell.

```
root@root/:$ python aalc -i examples/tuto0.aal -s
....
shell >
```

— Type help to show the shell help.

```
shell >help
  Shell Help
 - call(macro, args)    call a macro where /
                         *macro : is the name of the macro
                         *args : a list of string; <<ex :["'args1'",
                           "'args2'", ..."'argsN'"]>>
 - clauses()            show all declared clauses in the loaded aal
    program
 - macros()             show all declared macros in the loaded aal
    program
 - load(lib)            load the librarie lib
 - quit / q             exit the shell
 - help / h / man()     show this help
 - self                 the current compiler instance of the loaded aal
    program
 - aalprog              the current loaded aal program
 - man(arg)             print the help for the given arg
 - hs(module)           hotswaping : reload the module
 - r()                  hot-swaping the shell
```

— Here an example, we print all clauses in the AAL program.

```
shell> clauses()
css_policy alice_pref
```

— self variable refers to the compiler instance.

```
shell> self
<AALCompiler.AALCompilerListener object at 0x7f8b00ce8630>
```

— man can be called on any element, it will show its documentation.

```
shell> man(self)
printing manual for <class 'AALCompiler.AALCompilerListener'>
Manual for aal compiler visitor
 - Attributes
         -  aalprog      Get the AAL program instance
         -  file         The AAL source file
         -  libs         Show the loaded libraries
         -  libsPath     Print the standard lib path
 - Methods
         - load_lib(lib_name)    Load an aal file
         - clause(clauseId)      Lookup for clause cluaseId
         - show_clauses()        Show all clauses (names
         - get_clauses()         Get all clauses (objects)
         - get_macros()          Get all macros (objects)
```

— AAL program instance

```
shell> man(aalprog)
printing manual for <class 'AALMetaModel.m_aalprog'>

    AAL program class.
    Note that clauses and macros extend a declarable type, but are not
     in the declarations dict

    Attributes
        - clauses: a list that contains all program clauses
        - declarations: a dictionary that contains lists of typed
     declarations
        - comments: a list that contains program s comment
```

```
        - macros: a list that contains program s macros declarations
        - macroCalls: a list that contains program s comment
```

— Calling a macro

```
call("validate", ["'css_policy'", "'alice_pref'"])
------------------------- Monodic check -------------------------
Monodic check passed !
------------------------- Starting Validity check
     -------------------------
c1 : css_policy
c2 : alice_pref
----- Checking c1 & c2 consistency :
  -> Satisfiable
----- Checking c1 => c2 :
  -> Satisfiable
----- Checking ~(c1 => c2) :
  -> Satisfiable
:: Solving trigger

[VALIDITY] Formula is not valid !
------------------------- Validity check End -------------------------
```

— Defining a new macro

```
shell> self.new_macro("toto", ["p"], """print(p)""")
shell> call("toto", ["'4'"])
4
```

— hotswaping commands are used for debugging purpose only. r() command allows you to reload the shell without exiting it after source code modification.

— hs(module) reloading other modules after source code modification without exiting. ! IMORTANT : to use hotswaping properly you must enable it explicitly in aalc arguments `-d / -hotswap`,

BIBLIOGRAPHY

[AN08] Irem Aktug and Katsiaryna Naliuka. « ConSpec – A Formal Language for Policy Specification. » In: *Electronic Notes in Theoretical Computer Science*. Vol. 197. 2008, pp. 45–58. ISBN: 1571-0661.

[Ala+06] Eric Alata, Marc Dacier, Yves Deswarte, M. Kaaâniche, K. Kortchinsky, Vincent Nicomette, Van-Hau Pham, and Fabien Pouget. « Collection and analysis of attack data based on honeypots deployed on the Internet. » In: *Quality of Protection - Security Measurements and Metrics*. 2006, pp. 79–91. DOI: 10.1007/978-0-387-36584-8_7. URL: http://dx.doi.org/10.1007/978-0-387-36584-8_7.

[All+05] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. « Adding Trace Matching with Free Variables to AspectJ. » In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA: ACM, 2005, pp. 345–364. ISBN: 1-59593-031-0. DOI: 10.1145/1094811.1094839. URL: http://doi.acm.org/10.1145/1094811.1094839.

[And74] J. Anderson. *Computer Security Technology Planning Study*. Tech. rep. ESD-TR-7351. Hanscom, MA: Electronic Systems Division, Hanscom Air Force Base, 1974.

[Ard+09] Claudio A. Ardagna et al. *PrimeLife Policy Language*. http://www.w3.org/2009/policy-ws/papers/Trabelisi.pdf. 2009.

[Azr+15] Monir Azraoui, Kaoutar Elkhiyaoui, Melek Önen, Karin Bernsmed, Anderson Santana De Oliveira, and Jakub Sendor. « A-PPL: An Accountability Policy Language. » In: *Data Privacy Management, Autonomous Spontaneous Security, and Security Assurance: 9th International Workshop, DPM 2014, 7th International Workshop, SETOP 2014, and 3rd International Workshop, QASA 2014, Wroclaw, Poland, September 10-11, 2014. Revised Selected Papers*. Ed. by Joaquin Garcia-Alfaro, Jordi Herrera-Joancomartí, Emil Lupu, Joachim Posegga, Alessandro Aldini, Fabio Martinelli, and Neeraj Suri. Cham: Springer International Publishing, 2015, pp. 319–326. ISBN: 978-3-319-17016-9. DOI:

10.1007/978-3-319-17016-9_21. URL: http://dx.doi.org/10.1007/978-3-319-17016-9_21.

[BB13] Reza Babaee and Seyed Morteza Babamir. « Runtime Verification of Service-oriented Systems: A Well-rounded Survey. » In: *Int. J. Web Grid Serv.* 9.3 (Aug. 2013), pp. 213–267. ISSN: 1741-1106. DOI: 10.1504/IJWGS.2013.055699. URL: http://dx.doi.org/10.1504/IJWGS.2013.055699.

[BK98] Fahiem Bacchus and Froduald Kabanza. « Planning for temporally extended goals. » In: *Annals of Mathematics and Artificial Intelligence* 22.1 (1998), pp. 5–27. ISSN: 1573-7470. DOI: 10.1023/A:1018985923441. URL: http://dx.doi.org/10.1023/A:1018985923441.

[BO12] Alistair Barros and Daniel Oberle. *Handbook of Service Description: USDL and Its Methods*. Springer Publishing Company, Incorporated, 2012. ISBN: 1461418631, 9781461418634.

[BKM10] David Basin, Felix Klaedtke, and Samuel Müller. « Policy Monitoring in First-Order Temporal Logic. » In: *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–18. ISBN: 978-3-642-14295-6. DOI: 10.1007/978-3-642-14295-6_1. URL: http://dx.doi.org/10.1007/978-3-642-14295-6_1.

[BF11] Andreas Bauer and Yliès Falcone. « Decentralised LTL Monitoring. » In: *CoRR* abs/1111.5133 (2011). URL: http://arxiv.org/abs/1111.5133.

[BF12] Andreas Bauer and Yliès Falcone. « Decentralised LTL Monitoring. » In: *FM 2012: Formal Methods: 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*. Ed. by Dimitra Giannakopoulou and Dominique Méry. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 85–100. ISBN: 978-3-642-32759-9. DOI: 10.1007/978-3-642-32759-9_10. URL: http://dx.doi.org/10.1007/978-3-642-32759-9_10.

[BKV13] Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. « From Propositional to First-Order Monitoring. » In: *Runtime Verification: 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*. Ed. by Axel Legay and Saddek Bensalem. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 59–75. ISBN: 978-3-642-40787-1. DOI: 10.1007/978-3-642-40787-1_4. URL: http://dx.doi.org/10.1007/978-3-642-40787-1_4.

[BKV15] Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. « The ins and outs of first-order runtime verification. » English. In: *Formal Methods in System Design* 46.3 (2015), pp. 286–316. ISSN: 0925-9856. DOI: 10.1007/s10703-015-0227-2. URL: http://dx.doi.org/10.1007/s10703-015-0227-2.

[BLS10] Andreas Bauer, Martin Leucker, and Christian Schallhart. « Comparing LTL Semantics for Runtime Verification. » In: *J. Log. and Comput.* 20.3 (June 2010), pp. 651–674. ISSN: 0955-792X. DOI: 10.1093/logcom/exn075. URL: http://dx.doi.org/10.1093/logcom/exn075.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. « Runtime Verification for LTL and TLTL. » In: *ACM Trans. Softw. Eng. Methodol.* 20.4 (Sept. 2011), 14:1–14:64. ISSN: 1049-331X. DOI: 10.1145/2000799.2000800. URL: http://doi.acm.org/10.1145/2000799.2000800.

[BMB10] Moritz Y. Becker, Alexander Malkis, and Laurent Bussard. « A Practical Generic Privacy Language. » In: ed. by Somesh Jha and Anish Mathuria. Vol. 6503. ICISS 2010. Springer, 2010, pp. 125–139. ISBN: 978-3-642-17713-2.

[Ben+14] Walid Benghabrit, Hervé Grall, Jean-Claude Royer, and Mohamed Sellami. « Accountability for Abstract Component Design. » In: *EUROMICRO DSD/SEAA 2014*. Verona, Italy, Aug. 2014, pp. 213–220. DOI: 10.1109/SEAA.2014.68. URL: https://hal.inria.fr/hal-00987165.

[BA05] Travis D. Breaux and Annie I. Anton. « Deriving Semantic Models from Privacy Policies. » In: *Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY '05)*. Stockholm, Sweden, June 2005, pp. 67–76. DOI: 10.1109/POLICY.2005.12.

[BBF06] Julien Brunel, Jean-Paul Bodeveix, and Mamoun Filali. « A state/event temporal deontic logic. » In: DEON'06. Utrecht, The Netherlands: Springer-Verlag, 2006, pp. 85–100. ISBN: 3-540-35842-0, 978-3-540-35842-8. DOI: 10.1007/11786849_9.

[BCM14] Denis Butin, Marcos Chicote, and Daniel Le Métayer. « Strong Accountability: Beyond Vague Promises. » In: *Reloading Data Protection: Multidisciplinary Insights and Contemporary Challenges*. Springer, 2014, pp. 343–369. ISBN: 978-94-007-7540-4.

[BLM14] Denis Butin and Daniel Le Métayer. « Log Analysis for Data Protection Accountability. » In: *FM 2014: Formal Methods: 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*. Ed. by Cliff Jones, Pekka Pihla-

jasaari, and Jun Sun. Cham: Springer International Publishing, 2014, pp. 163–178. ISBN: 978-3-319-06410-9. DOI: 10.1007/978-3-319-06410-9_12. URL: http://dx.doi.org/10.1007/978-3-319-06410-9_12.

[BBG11] Rajkumar Buyya, James Broberg, and Andrzej M. Goscinski. *Cloud Computing Principles and Paradigms*. Wiley Publishing, 2011. ISBN: 9780470887998.

[Cas81] Hector-Neri Castañeda. « The Paradoxes of Deontic Logic: The Simplest Solution to all of them in one Fell Swoop. » In: *New Studies in Deontic Logic: Norms, Actions, and the Foundations of Ethics*. Ed. by Risto Hilpinen. Dordrecht: Springer Netherlands, 1981, pp. 37–85. ISBN: 978-94-009-8484-4. DOI: 10.1007/978-94-009-8484-4_2. URL: http://dx.doi.org/10.1007/978-94-009-8484-4_2.

[Ced+05] J.G. Cederquist, R. Corin, M.A.C. Dekker, S. Etalle, and J.I. Den Hartog. « An Audit Logic for Accountability. » In: *POLICY'05*. Ieee, 2005, pp. 34–43. DOI: 10.1109/POLICY.2005.5.

[CR09] Feng Chen and Grigore Roşu. « Parametric Trace Slicing and Monitoring. » In: *Tools and Algorithms for the Construction and Analysis of Systems: 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Ed. by Stefan Kowalewski and Anna Philippou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 246–261. ISBN: 978-3-642-00768-2. DOI: 10.1007/978-3-642-00768-2_23. URL: http://dx.doi.org/10.1007/978-3-642-00768-2_23.

[Che16] Ronan-Alexandre Cherrueau. « Un langage de composition des techniques de sécurité pour préserver la vie privée dans le nuage. (A Compositional Language of Security Techniques for Information Privacy in the Cloud). » PhD thesis. École des mines de Nantes, France, 2016. URL: https://tel.archives-ouvertes.fr/tel-01416166.

[Cho95] Jan Chomicki. « Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding. » In: *ACM Trans. Database Syst.* 20.2 (June 1995), pp. 149–186. ISSN: 0362-5915. DOI: 10.1145/210197.210200. URL: http://doi.acm.org/10.1145/210197.210200.

[Cho+13] Omar Chowdhury, Andreas Gampe, Jianwei Niu, Jeffery von Ronne, Jared Bennatt, Anupam Datta, Limin Jia, and William H. Winsborough. « Privacy promises that can be kept: a policy analysis method with application to the HIPAA privacy rule. » In: *SACMAT*. Ed. by Mauro Conti,

Jaideep Vaidya, and Andreas Schaad. ACM, 2013, pp. 3–14. ISBN: 978-1-4503-1950-8.

[Cra+13] Sjoerd Cranen, JanFriso Groote, JeroenJ.A. Keiren, FrankP.M. Stappers, ErikP. Vink, Wieger Wesselink, and TimA.C. Willemse. « An Overview of the mCRL2 Toolset and Its Recent Advances. » In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 7795. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 199–213.

[Dam+01] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. « The Ponder Policy Specification Language. » In: *POLICY*. Ed. by Morris Sloman, Jorge Lobo, and Emil Lupu. Vol. 1995. Lecture Notes in Computer Science. Springer, 2001, pp. 18–38. ISBN: 3-540-41610-2.

[DGV13] Giuseppe De Giacomo and Moshe Y. Vardi. « Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. » In: *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*. IJCAI '13. Beijing, China: AAAI Press, 2013, pp. 854–860. ISBN: 978-1-57735-633-2. URL: http://dl.acm.org/citation.cfm?id=2540128.2540252.

[DeY+10] Henry DeYoung, Deepak Garg, Limin Jia, Dilsun Kaynar, and Anupam Datta. « Experiences in the logical specification of the HIPAA and GLBA privacy laws. » In: *9th Annual ACM Workshop on Privacy in the Electronic Society (WPES '10)*. Chicago, Illinois, USA, 2010, pp. 73–82. DOI: 10.1145/1866919.1866930.

[Dir95] Directive, E. U. *Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data*. http://ec.europa.eu/justice/policies/privacy/docs/95-46-ce/dir1995-46_part1_en.pdf. 1995.

[Dix+07] Clare Dixon, Michael Fisher, Boris Konev, and Alexei Lisitsa. « Efficient First-Order Temporal Logic for Infinite-State Systems. » In: *CoRR* abs/cs/0702036 (2007).

[EU] EU. *GDPR*. http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32016R0679.

[EW07] Sandro Etalle and William H. Winsborough. « A posteriori compliance control. » In: *SACMAT 2007*. Ed. by Volkmar Lotz and Bhavani M. Thuraisingham. ACM, 2007, pp. 11–20. ISBN: 978-1-59593-745-2.

[Fei+12] Joan Feigenbaum, Aaron D. Jaggard, Rebecca N. Wright, and Hongda Xiao. *Systematizing "Accountability" in Computer Science*. Tech. rep. YALEU/DCS/TR-1452. University of Yale, 2012.

[Fel90] Fred Feldman. « A Simpler Solution to the Paradoxes of Deontic Logic. » In: *Philosophical Perspectives* 4 (1990), pp. 309–341. ISSN: 15208583, 17582245. URL: http://www.jstor.org/stable/2214197.

[FG+06] M. C. Fernández-Gago, U. Hustadt, C. Dixon, M. Fisher, and B. Konev. « First-Order Temporal Verification in Practice. » En. In: *Journal of Automated Reasoning* 34.3 (2006), pp. 295–321. ISSN: 1573-0670.

[Fis08] Michael Fisher. « Temporal Representation and Reasoning. » In: *Handbook of Knowledge Representation*. Ed. by Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter. Amsterdam: Elsevier, 2008, pp. 513–550.

[Fis11] Michael Fisher. *An Introduction to Practical Formal Methods using Temporal Logic*. Wiley, 2011. ISBN: 978-0-470-02788-2.

[GS12] Paul Gastin and Nathalie Sznajder. « Decidability of well-connectedness for distributed synthesis. » In: *Information Processing Letters* 112.24 (Dec. 2012), pp. 963–968. ISSN: 0020-0190 (print), 1872-6119 (electronic). DOI: http://dx.doi.org/10.1016/j.ipl.2012.08.018.

[GMM06] Alexandre Genon, Thierry Massart, and Cédric Meuter. « Monitoring Distributed Controllers: When an Efficient LTL Algorithm on Sequences is Needed to Model-check Traces. » In: *Proceedings of the 14th International Conference on Formal Methods*. FM'06. Hamilton, Canada: Springer-Verlag, 2006, pp. 557–572. ISBN: 3-540-37215-6, 978-3-540-37215-8. DOI: 10.1007/11813040_37. URL: http://dx.doi.org/10.1007/11813040_37.

[GP10] Alwyn Goodloe and Lee Pike. *Monitoring Distributed Real-Time Systems: A Survey and Future Directions*. 2010.

[Gro+08] J.F. Groote, A. Mathijssen, M.A. Reniers, Y.S. Usenko, and M.J. van Weerdenburg. « Process Algebra for Parallel and Distributed Processing. » In: ed. by M. Alexander and W. Gardner. Chapman and Hall, 2008. Chap. Analysis of distributed systems with mCRL2, pp. 99–128.

[GMS05] Carl A. Gunter, Michael J. May, and Stuart G. Stubblebine. « A Formal Privacy System and Its Application to Location Based Services. » In: *Privacy Enhancing Technologies: 4th International Workshop, PET 2004, Toronto, Canada, May 26-28, 2004. Revised Selected Papers*. Ed. by

David Martin and Andrei Serjantov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 256–282. ISBN: 978-3-540-31960-3. DOI: 10.1007/11423409_17. URL: http://dx.doi.org/10.1007/11423409_17.

[Hae+10] Andreas Haeberlen, Paarijaat Aditya, Rodrigo Rodrigues, and Peter Druschel. « Accountable Virtual Machines. » In: *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI*. 2010, pp. 119–134. ISBN: 978-1-931971-79-9.

[HV08] S. Halle and R. Villemaire. « Runtime Monitoring of Message-Based Workflows with Data. » In: *Enterprise Distributed Object Computing Conference, 2008. EDOC '08. 12th International IEEE*. 2008, pp. 63–72. DOI: 10.1109/EDOC.2008.32.

[HPT07] Jörg Hansen, Gabriella Pigozzi, and Leendert van der Torre. « Ten Philosophical Problems in Deontic Logic. » In: *Normative Multi-agent Systems*. Ed. by Guido Boella, Leon van der Torre, and Harko Verhagen. Dagstuhl Seminar Proceedings 07122. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. URL: http://drops.dagstuhl.de/opus/volltexte/2007/941.

[Hea96] Health Resources and Services Administration. *Health Insurance Portability and Accountability Act*. 1996.

[Hea] Heartbleed. *Heartbleed*. https://fr.wikipedia.org/wiki/Heartbleed.

[Hil+07] Manuel Hilty, Alexander Pretschner, David A. Basin, Christian Schaefer, and Thomas Walter. « A Policy Language for Distributed Usage Control. » In: *ESORICS*. Ed. by Joachim Biskup and Javier Lopez. Vol. 4734. Lecture Notes in Computer Science. Springer, 2007, pp. 531–546. ISBN: 978-3-540-74834-2.

[HWZ00] Ian M. Hodkinson, Frank Wolter, and Michael Zakharyaschev. « Decidable fragment of first-order temporal logics. » In: *Ann. Pure Appl. Logic* 106.1-3 (2000), pp. 85–134.

[Jag+09] Radha Jagadeesan, Alan Jeffrey, Corin Pitcher, and James Riely. « Towards a theory of accountability and audit. » In: ESORICS'09. Saint-Malo, France: Springer-Verlag, 2009, pp. 152–167.

[KP06] Raman Kazhamiakin and Marco Pistore. « Analysis of Realizability Conditions for Web Service Choreographies. » In: *Formal Techniques for Networked and Distributed Systems - FORTE 2006, 26th IFIP WG 6.1 International Conference, Paris, France, September 26-29, 2006*. Ed. by

Elie Najm, Jean-François Pradat-Peyre, and Véronique Donzeau-Gouge. Vol. 4229. Lecture Notes in Computer Science. Springer, 2006, pp. 61–76. ISBN: 3-540-46219-8. URL: http://dx.doi.org/10.1007/11888116_5.

[KL03] Shawn Kerrigan and Kincho H. Law. « Logic-Based Regulation Compliance-Assistance. » In: *International Conference on Artificial Intelligence and Law*. 2003, pp. 126–135.

[LSE03] D. Davide Lamanna, James Skene, and Wolfgang Emmerich. « SLAng: A Language for Defining Service Level Agreements. » In: *Proceedings of the The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems*. FTDCS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 100–. ISBN: 0-7695-1910-5.

[LS95] F. Laroussinie and Ph. Schnoebelen. « A hierarchy of temporal logics with past. » In: *Theoretical Computer Science* 148.2 (1995), pp. 303–324.

[Lau17] Laure LANDES-GRONOWSKI. *Le Règlement Général sur la Protection des Données (RGPD)*. http://www.avistem.com. 2017.

[LH10] Michel Ludwig and Ullrich Hustadt. « Implementing a fair monodic temporal logic prover. » In: *AI Commun* 23.2-3 (2010), pp. 69–96.

[MKL09] Tim Mather, Subra Kumaraswamy, and Shahed Latif. *Cloud Security and Privacy: An Enterprise Perspective on Risks and Compliance*. O'Reilly Media, Inc., 2009. ISBN: 0596802765, 9780596802769.

[MGL06] M. J. May, C. A. Gunter, and I. Lee. « Privacy APIs: access control techniques to analyze and verify legal privacy policies. » In: *19th IEEE Computer Security Foundations Workshop (CSFW'06)*. 2006, 13 pp.–97. DOI: 10.1109/CSFW.2006.24.

[McN10] Ashley T. McNeile. « Protocol contracts with application to choreographed multiparty collaborations. » In: *Service Oriented Computing and Applications* 4.2 (2010), pp. 109–136. URL: http://dx.doi.org/10.1007/s11761-010-0060-9.

[Mét09] Daniel Le Métayer. « A Formal Privacy Management Framework. » In: *Formal Aspects in Security and Trust: 5th International Workshop, FAST 2008 Malaga, Spain, October 9-10, 2008 Revised Selected Papers*. Ed. by Pierpaolo Degano, Joshua Guttman, and Fabio Martinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 162–176. ISBN: 978-3-642-01465-9. DOI: 10.1007/978-3-642-01465-

9_11. URL: https://doi.org/10.1007/978-3-642-01465-9_11.

[MÓ9] Daniel Le Métayer. « A formal privacy management framework. » In: *Formal Aspects in Security and Trust* (2009), pp. 1–15.

[Mic] Michael Ludwig. *TsPass*. http://lat.inf.tu-dresden.de/~michel/software/tspass.

[MS03] Kevin D. Mitnick and William L. Simon. *The Art of Deception: Controlling the Human Element of Security*. New York, NY, USA: John Wiley & Sons, Inc., 2003. ISBN: 076454280X.

[MS06] Kevin D. Mitnick and William L. Simon. *The art of intrusion*. Indianapolis., 2006. ISBN: 0764589423.

[OAS13] OASIS Standard. *eXtensible Access Control Markup Language (XACML) Version 3.0. 22 January 2013*. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html. 2013.

[OAS06] OASIS Web Service Security (WSS) TC. *Web Services Security: SOAP Message Security 1.1*. https://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf. 2006.

[OAS12] OASIS Web Services Secure Exchange (WS-SX) TC. *WS-Trust 1.4*. http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/errata01/os/ws-trust-1.4-errata01-os-complete.html. 2012.

[PW13] Siani Pearson and Nick Wainwright. « An interdisciplinary approach to accountability for future internet service provision. » In: *International Journal of Trust Management in Computing and Communications* 1.1 (2013), pp. 52–72. DOI: 10.1504/IJTMCC.2013.052524.

[PD11] Guillaume Piolle and Yves Demazeau. « Representing privacy regulations with deontico-temporal operators. » In: *Web Intelligence and Agent Systems* 9.3 (2011), pp. 209–226.

[Pnu77] Amir Pnueli. « The Temporal Logic of Programs. » In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. SFCS '77. Washington, DC, USA: IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32. URL: http://dx.doi.org/10.1109/SFCS.1977.32.

[RS14] R. Ramanujam and S. Sheerazuddin. « A Local Logic for Realizability in Web Service Choreographies. » In: *Proceedings 10th International Workshop on Automated Specification and Verification of Web Systems, WWV 2014, Vienna, Austria, July 18, 2014*. Ed. by Maurice H. ter Beek and António Ravara. Vol. 163. EPTCS. 2014, pp. 16–35. URL: http://dx.doi.org/10.4204/EPTCS.163.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004. ISBN: 0321245628.

[Sch99] Andreas Schedler. *Self-Restraining State: Power and Accountability in New Democracies*. Boulder, Colo.: Lynne Rienner Publishers, 1999.

[SS14] T. Scheffel and M. Schmitz. « Three-valued asynchronous distributed runtime verification. » In: *Formal Methods and Models for Codesign (MEMOCODE), 2014 Twelfth ACM/IEEE International Conference on*. 2014, pp. 52–61. DOI: 10.1109/MEMCOD.2014.6961843.

[Sch+08] Matthew A. Scholl, Kevin M. Stine, Joan Hash, Pauline Bowen, L. Arnold Johnson, Carla Dancy Smith, and Daniel I. Steinberg. *SP 800-66 Rev. 1. An Introductory Resource Guide for Implementing the Health Insurance Portability and Accountability Act (HIPAA) Security Rule*. Tech. rep. Gaithersburg, MD, United States, 2008.

[Sch12] Viktor Schuppan. « Towards a notion of unsatisfiable and unrealizable cores for LTL. » In: *Science of Computer Programming* 77.7–8 (July 2012), pp. 908–939. ISSN: 0167-6423 (print), 1872-7964 (electronic). DOI: http://dx.doi.org/10.1016/j.scico.2010.11.004.

[SD11] Viktor Schuppan and Luthfi Darmawan. « Evaluating LTL Satisfiability Solvers. » In: *ATVA*. Ed. by Tevfik Bultan and Pao-Ann Hsiung. Vol. 6996. LNCS. Springer, 2011, pp. 397–413. ISBN: 978-3-642-24371-4.

[SRA03] Koushik Sen, Grigore Roşu, and Gul Agha. « Generating Optimal Linear Temporal Logic Monitors by Coinduction. » In: *Advances in Computing Science – ASIAN 2003. Progamming Languages and Distributed Computation Programming Languages and Distributed Computation: 8th Asian Computing Science Conference, Mumbai, India, December 10-12, 2003. Proceedings*. Ed. by Vijay A. Saraswat. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 260–275. ISBN: 978-3-540-40965-6. DOI: 10.1007/978-3-540-40965-6_17. URL: https://doi.org/10.1007/978-3-540-40965-6_17.

[Sen+04] Koushik Sen, A. Vardhan, Gul Agha, and G. Rosu. « Efficient decentralized monitoring of safety in distributed systems. » In: *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*. 2004, pp. 418–427. DOI: 10.1109/ICSE.2004.1317464.

[Sen+06] Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Roşu. « Decentralized Runtime Analysis of Multithreaded Applications. » In: *NSF Next Generation Software Program Workshop (NSFNGS'06) (Satellite Workshop of IPDPS'06)*. (To Appear). IEEE Digital Library, 2006.

[She+14] Yan Shen, Jianwen Li, Zheng Wang, Ting Su, Bin Fang, Geguang Pu, Wanwei Liu, and Mingsong Chen. « Runtime Verification by Convergent Formula Progression. » In: *Proceedings of the 2014 21st Asia-Pacific Software Engineering Conference - Volume 01*. APSEC '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 255–262. ISBN: 978-1-4799-7426-9. DOI: 10.1109/APSEC.2014.47. URL: http://dx.doi.org/10.1109/APSEC.2014.47.

[Sto10] Volker Stolz. « Temporal Assertions with Parametrized Propositions*. » In: *J. Log. and Comput.* 20.3 (June 2010), pp. 743–757. ISSN: 0955-792X. DOI: 10.1093/logcom/exn078. URL: http://dx.doi.org/10.1093/logcom/exn078.

[SSL12] S. Sundareswaran, A. Squicciarini, and D. Lin. « Ensuring Distributed Accountability for Data Sharing in the Cloud. » In: *Dependable and Secure Computing, IEEE Transactions on* 9.4 (2012), pp. 556–568.

[TDW13] Michael Carl Tschantz, Anupam Datta, and Jeannette M. Wing. « Purpose Restrictions on Information Use. » In: ed. by Jason Crampton, Sushil Jajodia, and Keith Mayes. Vol. 8134. ESORICS 2013. Springer, 2013, pp. 610–627. ISBN: 978-3-642-40202-9.

[US] US. *GLBA*. https://www.law.cornell.edu/uscode/text/15/chapter-94/subchapter-I.

[P3p] *W3C. Platform for Privacy Preferences. http://www.w3.org/P3P/.*

[Wei+09] Wei Wei, Juan Du, Ting Yu, and Xiaohui Gu. « SecureMR: A Service Integrity Assurance Framework for MapReduce. » In: *Proceedings of the 2009 Annual Computer Security Applications Conference*. ACSAC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 73–82. ISBN: 978-0-7695-3919-5. DOI: 10.1109/ACSAC.2009.17.

[Wei+08] Daniel J. Weitzner, Harold Abelson, Tim Berners-Lee, Joan Feigenbaum, James Hendler, and Gerald Jay Sussman. « Information accountability. » In: *Commun. ACM* 51.6 (June 2008), pp. 82–87. ISSN: 0001-0782. DOI: 10.1145/1349026.1349043.

[Zha+10] L. Zhao, T. Tang, J. Wu, and T. Xu. « Runtime Verification with Multi-valued Formula Rewriting. » In: *Theoretical Aspects of Software Engineering (TASE), 2010 4th IEEE International Symposium on*. 2010, pp. 77–86. DOI: 10.1109/TASE.2010.13.

[ZX12] Yang Xiao Zhifeng Xiao Nandhakumar Kathiresshan. « A survey of accountability in computer networks and distributed systems. » In: *Security and Communication Networks* (2012).

[ZWL10] Joe Zou, Yan Wang, and Kwei-Jay Lin. « A Formal Service Contract Model for Accountable SaaS and Cloud Services. » In: IEEE, 2010, pp. 73–80. DOI: 10.1109/SCC.2010.85.

Colophon

This document was written using LaTeX. This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede.

https://bitbucket.org/amiede/classicthesis/

The graphics and illustrations were made using yEd Graph Editor, inkscape and Gimp.

https://www.yworks.com/yed
https://inkscape.org
https://www.gimp.org

Developed tools and softwares presented in this thesis are released under GNU GPLv3.

https://github.com/hkff/AccLab
https://github.com/hkff/FodtlMon
https://github.com/hkff/PyMon
https://github.com/hkff/AccMon

UNIVERSITE BRETAGNE LOIRE

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Thèse de Doctorat

Walid BENGHABRIT

UN MODEL FORMEL POUR LA RESPONSABILISATION

A FORMAL MODEL FOR ACCOUNTABILITY

Résumé

Nous assistons à la démocratisation des services du cloud et de plus en plus d'utilisateurs (individuels ou entreprises) utilisent ces services dans la vie de tous les jours. Dans ces scénarios, les données personnelles transitent généralement entre plusieurs entités. L'utilisateur final se doit d'être informé de la collecte, du traitement et de la rétention de ses données personnelles, mais il doit aussi pouvoir tenir pour responsable le fournisseur de service en cas d'atteinte à sa vie privée. La responsabilisation (ou accountability) désigne le fait qu'un système ou une personne est responsable de ses actes et de leurs conséquences. Dans cette thèse nous présentons un framework de responsabilisation AccLab qui permet de prendre en considération la responsabilisation dès la phase de conception d'un système jusqu'à son implémentation. Afin de réconcilier le monde juridique et le monde informatique, nous avons développé un langage dédié nommé AAL permettant d'écrire des obligations et des politiques de responsabilisation. Ce langage est basé sur une logique formelle FOTL ce qui permet de vérifier la cohérence des politiques de responsabilisation ainsi que la compatibilité entre deux politiques. Les politiques sont ensuite traduites en une logique temporelle distribuée que nous avons nommée FO-DTL 3, cette dernière est associée à une technique de monitorage basée sur la réécriture de formules. Enfin nous avons développé un outil monitorage appelé AccMon qui fournit des moyens de surveiller les politiques de responsabilisation dans le contexte d'un système réel. Les politiques sont fondées sur la logique FO-DTL 3 et le framework peut agir en mode centralisée ou distribuée et fonctionne à la fois en ligne et hors ligne.

Mots clés

Responsabilisation, sécurité, vie privée, logique temporelle, vérification, monitorage.

Abstract

Nowadays we are witnessing the democratization of cloud services. As a result, more and more end-users (individuals and businesses) are using these services in their daily life. In such scenarios, personal data is generally flowed between several entities. End-users need to be aware of the management, processing, storage and retention of personal data, and to have necessary means to hold service providers accountable for the use of their data. In this thesis we present an accountability framework called Accountability Laboratory (AccLab) that allows to consider accountability from design time to implementation time of a system. In order to reconcile the legal world and the computer science world, we developed a language called Abstract Accountability Language (AAL) that allows to write obligations and accountability policies. This language is based on a formal logic called First Order Linear Temporal Logic (FOTL) which allows to check the coherence of the accountability policies and the compliance between two policies. These policies are translated into a temporal logic called FO-DTL 3, which is associated with a monitoring technique based on formula rewriting. Finally, we developed a monitoring tool called Accountability Monitoring (AccMon) which provides means to monitor accountability policies in the context of a real system. These policies are based on FO-DTL 3 logic and the framework can act in both centralized and distributed modes and can run into on-line and off-line modes.

Keywords

Accountability, security, privacy, temporal logic, verification, monitoring.