UNIVERSITY OF MALAGA

Trust Engineering Framework for Software Services

by

Francisco Moyano Lara
Computer Science and Languages Department
University of Malaga

submitted in fulfillment of the requirements for the
Degree of Doctor in Computer Science

Advisors

M. Carmen Fernández Gago
Senior Research Fellow at University of Malaga
Fco. Javier López Muñoz
Full Professor at University of Malaga

June 2015

AUTOR: Francisco Moyano Lara
http://orcid.org/0000-0002-0148-9624

EDITA: Publicaciones y Divulgación Científica. Universidad de Málaga

A Noni y mis padres, las personas más importantes en mi vida.

Acknowledgements

First of all, I would like to express my deepest gratitude to my supervisors, Carmen Fernandez and Javier Lopez. Both of them have made the process of developing my thesis smooth and rewarding. And more important, over time they have become more than supervisors and colleagues; they have become really good friends, something I feel proud of because they are excellent persons.

Along the development of my thesis, I have met and worked with lots of people at NICS Lab. There are not enough kind words in the English vocabulary that I can devote to these people. Guys, you are awesome and my experience working with you has been legen... wait for it!

I must start expressing my gratitude to my office buddies and Pokemon friends, Ana and David, with whom I have shared so many epic moments. I still recall my first work in the group together with Pablo, a fantastic person who initiated me in the research world, concretely with RFID readers and tags (which by the way may have inclined me towards becoming the tags master...). My next Pokemon master was Rodrigo, an authentic geek (in the best sense of the word) with whom I share the passion for videogames and sushi. He made a fantastic job at co-supervising my master thesis.

The rest of NICS guys are equally awesome, including those not working here any longer. I can only hold good thoughts and share kind words about every of them: Gerardo, Noelia, Rubén, Lorena, Jesús, Isaac, Cristina, Dani, Miguel, Saúl, Jose, Edu, Pepe, Montes, Onieva... All of them have contributed something to me, and have made me feel really appreciated. Thank you!

There are three people that have been essential during my thesis, but much more importantly, over my life: my parents, Ana and Francis, and my girlfriend Noni. Regarding my parents, I can only say that they are always there for me and that they are the best parents in the world, and that without them, I would not have come this far.

As for Noni, what can I say? She is one of the best persons I have ever known and our love has always inspired and encouraged me to go ahead and become a better person. I am always thrilled when I think of our future together.

The good thing of having a small yet loving family is that I can share even more love with each of its members: my grandparents, Conchi, Paco, Chus and Miguel (who, in spite of not being physically here any longer, I know that he always cares for me); my aunts and uncles, Suli, Miguel, Chari and Sergio; my cousins Ali, Sergio, Dani and Miguelito; my mother-in-law Manoli; and my brother-in-law Juanma. I have been so lucky to count on all of them...

Some long-life friends must be listed here of course. Ernesto, who was my first best friend and who taught me how to ride a bike; Raul, with whom I learned how to dance Latin music; Oliver, the best musician I will ever know; Jose Angel (brown), with whom I share the most exciting discussions about religion; Jose Angel (blonde), who helped me discover how Cadiz and Jerez can bring good friends together even more; Santi, because deep friendship links can never be broken, even when distance might seem an unavoidable obstacle.

I would also like to devote some thankful words to my colleagues during the master's degree, especially those in the ISP group, who have become close friends over time and with whom I have shared so many fun and geek moments.

During the development of NESSoS, the project that helped shape my thesis, I have met clever and nice people with whom I had the opportunity to work. Kristian, a great German guy who will always have a place in Malaga; Jorge Cuellar, who provided me with useful insights that helped me improve my research; Benoit, who gave me a warm welcome to his fantastic research lab during my ph.D. stay in Rennes; and Jean-Emile, who worked closely with me during my stay and helped me every time I found a stumbling block.

I am also grateful to some institutions and projects for their funding and support. This thesis has been primarily funded by the Spanish Ministry of Education under the FPU fellowship. Special thanks go to the NESSoS (FP7 256890) project, because, as mentioned earlier, it has helped me pave the path of my thesis, it has given me the opportunity to collaborate with other institutions and high quality researchers, as well as to receive feedback for improving the quality of my research and make good friends.

Agradecimientos

Antes de nada, quisiera expresar mi más profundo agradecimiento a mis directores, Carmen Fernández y Javier López. Ambos han convertido el desarrollo de la tesis en un proceso agradable y gratificante. Y lo que es más importante, a lo largo del tiempo se han convertido en más que directores y compañeros; se han convertido en buenos amigos, lo cual me llena de orgullo porque son unas personas fantásticas.

A lo largo del desarrollo de mi tesis, he conocido y trabajado con muchas personas en NICS Lab. No hay suficientes palabras amables que pueda dedicar a estas personas. Sois todos geniales, y mi experienca trabajando con vosotros ha sido legen... ¡espera un momento!

Debo empezar por mostrar mi gratitud a mis compis de oficina y amigos Pokemon, Ana y David, con los cuales he vivido momentos realmente épicos. Aún recuerdo mi primer trabajo en el grupo de la mano de Pablo, una persona fantástica que me inició en el mundo de la investigación, concretamente con etiquetas y lectores RFID (lo que pensándolo bien puede que haya llevado a convertirme en el maestro de las etiquetas...). Mi siguiente maestro Pokemon fue Rodrigo, un verdadero friki (en el mejor sentido de la palabra), con quién comparto la pasión por los videojuegos y el sushi, y quien hizo un trabajo sobresaliente al co-dirigir mi proyecto final de carrera.

El resto de los miembros de NICS son igualmente alucinantes, incluyendo los que ya no trabajan aquí. Sólo puedo compartir buenas palabras sobre cada uno de ellos: Gerardo, Noelia, Rubén, Lorena, Jesús, Isaac, Cristina, Dani, Miguel, Saúl, Jose, Edu, Pepe, Montes, Onieva... Todos ellos me han aportado algo, y me han hecho sentir realmente querido en el grupo. ¡Gracias!

Hay tres personas que han sido imprescindibles durante esta tesis, pero mucho más importante, a lo largo de mi vida: mis padres Ana y Francis, y mi novia Noni. En cuanto a mis padres, sólo puedo decir que siempre están ahí para mí y que son los

mejores padres del mundo, y que sin ellos, no habría llegado tan lejos. Y de Noni, ¿qué puedo decir? Es una de las mejores personas que he conocido y nuestro amor siempre me ha inspirado y animado a seguir adelante y a convertirme en mejor persona. Me llena de emoción y alegría pensar en nuestro futuro juntos.

Lo bueno de tener una familia pequeña es que puedo compartir incluso más amor con cada uno de sus miembros: mis abuelos, Conchi, Paco, Chus y Miguel (quién, a pesar de no estar físicamente, sé que siempre está cuidando de mí); mis tías y tíos, Suli, Miguel, Chari y Sergio; mis primos Ali, Sergio, Dani y Miguelito; la familia de Noni, Manoli y Juanma. He sido tan afortunado de poder contar con todos ellos...

Mis amigos de toda la vida tienen un lugar aquí por supuesto. Ernesto, que fue mi primer mejor amigo y quién me enseñó a montar en bici; Raúl, de quien aprendí a bailar música latina; Óliver, el mejor músico que jamás conoceré; José Ángel (moreno), con quien he compartido las discusiones más interesantes sobre religión; José Ángel (rubio), con quién he descubierto que Cádiz y Jerez pueden unir a los buenos amigos incluso más; Santi, porque la auténtica amistad nunca pueden romperse, incluso cuando la distancia pueda parecer un obstáculo insalvable.

También me gustaría dedicar unas palabras de agradecimiento a mis compañeros durante la carrera, especialmente a aquéllos del grupo ISP, quienes se han convertido en buenos amigos a lo largo del tiempo y con quienes he compartido tantos momentos divertidos y frikis.

Durante el desarrollo de NESSoS, el proyecto que ha ayudado a dar forma a mi tesis, he conocido gente amable e inteligente con quién he tenido la oportunidad de trabajar. Kristian, un genial chico alemán que siempre tendrá un lugar en Málaga; Jorge Cuéllar, quién me ha ofrecido interesantes reflexiones para mejorar mi investigación; Benoit, que me acogió amablemente en su genial grupo de investigación durante mi estancia en Rennes; y Jean-Emile, con quién trabajé estrechamente durante mi estancia y que me ayudó cada vez que me encontraba una piedra en el camino.

Esta tesis ha sido principalmente financiada por el Ministerio de Educación a través de la beca FPU. Debo agradecer especialmente al proyecto NESSoS (FP7 256890) porque ha pavimentado el camino de mi tesis, me ha dado la oportunidad de colaborar con otras instituciones e investigadores de alto prestigio, así como de recibir opiniones para mejorar mi investigación y hacer buenos amigos.

Contents

List of Figures **xi**

List of Tables **xv**

Listings **xviii**

Acronyms **xix**

1 Introduction **1**

1.1 Research Scope 1

1.1.1 Secure System Engineering 2

1.1.2 Trust Management 3

1.1.3 Trust, Security, Risk and Trustworthiness 5

1.2 Goals and Organization 8

1.2.1 Thesis Contributions 9

1.2.2 Thesis Outline 10

1.3 Publications and Funding 14

2 Understanding Trust: A Systematic Analysis **17**

2.1 Literature Review 18

2.1.1 Trust Taxonomies 18

2.1.2 Trust in the Computing Domain 21

2.1.3 Trust in the Software Development Life Cycle 25

2.2 Trust Conceptual Model 32

2.2.1 Trust Definitions 32

2.2.2 Trust Models: Definition and Classification 34

2.2.3 Trust Models Concepts 36

2.2.4 Comparison Framework: A Case Study 42

3 Incorporating Trust Engineering in Early Phases of the SDLC 47

3.1 Trust-supported Cloud Sourcing Decision in the Planning Phase 48
3.1.1 Trust Evaluation in the Cloud 49
3.1.2 Trust-Aware Methodology 52
3.1.3 Application Example: eHealth 59
3.1.4 Discussion 63
3.2 Trust-supported Threats Analysis 65
3.2.1 The SI* Framework 67
3.2.2 Asset Model 69
3.2.3 Trust Model 70
3.2.4 Threat Model 74
3.2.5 Application Example: eHealth 75
3.2.6 Discussion 83
3.3 Eliciting and Representing Trust and Reputation Requirements 84
3.3.1 Problem Frames 84
3.3.2 Trust Extensions to UML4PF 85
3.3.3 Formal Checking of Trust 88
3.3.4 Application Example: Smart Grid 91
3.3.5 Discussion 98
3.4 Designing Trust and Reputation Solutions 101
3.4.1 Use Case Diagram 102
3.4.2 Class Diagram 103
3.4.3 Deployment Diagram 104
3.4.4 Application Example: eHealth 105
3.4.5 Discussion 112

4 Enabling Trust and Reputation during Implementation 113

4.1 Framework Requirements 114
4.2 High-Level Architecture 115
4.2.1 Model Layer 116
4.2.2 Relational Layer 117
4.2.3 Computation Layer 117

4.2.4 User-Defined Layer . . . 118
4.3 Low-Level Architecture . . . 118
4.3.1 Components Decomposition . . . 120
4.3.2 Data Structures . . . 122
4.3.3 Incorporating Trustor's Subjective Factors . . . 124
4.4 Implementation Guidelines . . . 124
4.4.1 Context . . . 125
4.4.2 Database Tables . . . 125
4.4.3 Messaging Infrastructure . . . 128
4.4.4 Engines . . . 128
4.4.5 Deployment . . . 130
4.5 Application Example: Social Cloud . . . 130
4.5.1 Scenario Description . . . 131
4.5.2 Trust Requirements . . . 131
4.5.3 Implementation . . . 132
4.6 Discussion . . . 139

5 Enabling Trust and Reputation at Runtime 143
5.1 Kevoree: A Models@run.time Development Platform . . . 144
5.1.1 Kevoree Development Framework . . . 146
5.1.2 Deployment in Kevoree . . . 148
5.2 Integrating Trust and Reputation in Models@Run.time . . . 148
5.2.1 Trust and Reputation Metamodels . . . 150
5.2.2 Trust Framework . . . 152
5.2.3 Reputation Framework . . . 157
5.3 Trust-based Self-Adaptation . . . 161
5.3.1 Policy-based Reconfiguration . . . 161
5.3.2 Implementation . . . 163
5.4 Application Example: A Trust-Aware Distributed Chat . . . 165
5.4.1 eBay Model . . . 165
5.4.2 Marsh's Trust Model . . . 168
5.4.3 PeerTrust . . . 172
5.4.4 REGRET . . . 176

5.5 Experimenal Results 179
5.6 Discussion 181

6 Conclusions **183**

A Resumen en español **193**
A.1 Marco de la tesis y objetivos 193
A.2 Resumen de contribuciones 195
A.3 Marco de trabajo conceptual de confianza 196
A.4 Evaluación de confianza de proveedores de cloud 200
A.5 Uso de confianza durante la fase de análisis de seguridad 202
A.5.1 Identificación de amenazas internas guiada por relaciones de confianza 202
A.5.2 Recogida de requisitos de confianza y reputación 204
A.6 Especificación de modelos de confianza y reputación 205
A.7 Marco de trabajo para la implementación de modelos de confianza y reputación 207
A.8 Marco de trabajo para la implementación de sistemas autoadaptativos en función de valores de confianza y reputación 210

References **213**

List of Figures

1.1 Secure System Development Life Cycle 2
1.2 Main Contributions in relation to the SDLC 11

2.1 Concepts Cloud for Trust Definitions 34
2.2 Common Concepts for Trust Models 37
2.3 Concepts for Decision Models 38
2.4 Concepts for Evaluation Models 41
2.5 General Framework for Trust Models Comparison 42

3.1 Overview of the Methodology 52
3.2 Comparison of Trust Intervals for the selected Cloud Providers 64
3.3 Contrasting Trust Thresholds and Trust Intervals 65
3.4 Threats Severity Levels 75
3.5 Patient Monitoring Scenario in SI* 78
3.6 A Trust Extension of the UML4PF Profile 86
3.7 Methodology Proposed for Engineering Trust and Reputation Requirements into Systems 92
3.8 Context Diagram for the Smart Metering Gateway 93
3.9 Domain Knowledge Diagram - Analysing the Trust Relationship Consumer-CLS 94
3.10 Domain Knowledge Diagram - Analysing Reputation Information 94
3.11 Domain Knowledge Diagram - Refinement of Trust and Reputation Information 95
3.12 Problem Diagram - Describing Trust and Reputation Engines 96
3.13 Sequence Diagram - Describing Trust and Reputation Events 97

3.14 Tool-Supported Modelling of Trust-Aware UML4PF 100
3.15 Tool-Supported OCL Evaluation 100
3.16 Use Case Diagram 106
3.17 Trust-aware Use Case Diagram 107
3.18 Patient-Physician relationship 108
3.19 Data Retrieval Trust Relationship 109
3.20 Quality Feedback Claim 109
3.21 Trust-aware Use Case Diagram (2nd Iteration) 110
3.22 Activity Diagram for Use Case *Set Physician Preferences* 111
3.23 Activity Diagram for Use Case *Ask for Second Opinion* 111
3.24 Trust-aware Deployment Diagram 112

4.1 The Framework in Context 114
4.2 High-Level Architecture of the Framework 116
4.3 Framework Architecture: Component Diagram 119
4.4 Framework Architecture: Sequence Diagram 120
4.5 Engine Dispatcher Component 121
4.6 Event Handler Component 121
4.7 Data Manager Component 122
4.8 Important Data Structures 123
4.9 Query API Call Sequence Diagram 125
4.10 Two Deployment Configurations 130
4.11 Social Cloud Scenario 132
4.12 Detailed Sequence Diagram of *sendEvent* API Call 136
4.13 Detailed Sequence Diagram of *sendEvent* API Call (cont.) 136

5.1 Kevoree Architectural Elements 145
5.2 Adaptation Process 146
5.3 Kevoree editor with three nodes 149
5.4 Kevscript Instructions 149
5.5 Trust Metamodel 150
5.6 Reputation Metamodel 151
5.7 Execution Time (measured in microseconds) 180

A.1 Ciclo de vida del desarrollo de sistemas seguros 194
A.2 Contribuciones 195

LIST OF FIGURES

List of Tables

2.1 Trust Definitions 33
2.2 Common Features Comparison 43
2.3 Decision Models Comparison 43
2.4 Evaluation Models Comparison 44
2.5 Evaluation Models Comparison (II) 44

3.1 Stakeholder Trust Template 54
3.2 Cloud Provider Trust Template 55
3.3 CSA Cloud Threats and their Evaluation 56
3.4 Trust Interval Summations 58
3.5 Trust Factors Quantification for Microsoft 62
3.6 Trust Thresholds 63
3.7 Trust Intervals for Cloud Providers 63
3.8 Permissions on Resource 68
3.9 Relationships between Resources 69
3.10 Predicates for ASP SI* Formalization 70
3.11 Predicates for the Asset Model 71
3.12 Axioms for Identifying Indirect Assets 72
3.13 Axioms for Identifying Insider Threats 76
3.14 ASP Formalization for SI* Model 79
3.15 ASP Rules for SI* Model Instantiation 80
3.16 OCL Expressions that support Trust-based Security Reasoning and Consistency Checks 91
3.17 Use Case Diagram Extensions 102
3.18 Class Diagram Extensions 103

3.19 Tagged Values for Class Diagrams 104
3.20 Deployment Diagram Extensions 105

4.1 Entity Table 126
4.2 Trust Relationship Table 126
4.3 Reputation Statement Table 127
4.4 Beliefs Table 127
4.5 Entity Table Example 137
4.6 Trust Relationship Table Example 138
4.7 Reputation Statement Table Example 138

5.1 Trust Factors for Marsh's Model 170
5.2 Amount of Framework-related Activities 180

Listings

3.1 IDHE001. List all biddable domains that are not a human entity 89
3.2 CCTV001. Check that all dependencies with a trust relationship have a dependency to a TrustValue 89
3.3 CCL001. List all sources of claims that are not a Human Entity 97
4.1 Engine Configuration 128
4.2 Excerpt of EngineManager 129
4.3 Creating a Bounded Claim 133
4.4 Binding Claim Type and Event 133
4.5 Implementing Trust and Reputation Engines 134
4.6 Configuring Engines 135
4.7 API Call for Sending the Event 135
4.8 Changing a Belief 138
4.9 Framework Hooks 139
5.1 Definition of Console in Kevoree 146
5.2 Querying the model@runtime layer 147
5.3 TrustEntity Component 152
5.4 Trust Entities Initialization 154
5.5 Adding Trust Relationships with the EMF API 155
5.6 TrustModel Component 156
5.7 CentralReputableEntity Component 157
5.8 ReputationManager Component 158
5.9 DistReputableEntity Component and Reputation Engine Initialization . 159
5.10 ReputationEngine Class 160
5.11 Reconfiguration Rules Processing 163
5.12 ScriptEngine: Remove Component 164

5.13 Console in the Ebay Reputation Model 166
5.14 Ebay Reputation Engine 167
5.15 Console in Marsh Trust Model 170
5.16 Trust Engine for Marsh's Model 172
5.17 Binding Reputation Engine and Console 174
5.18 Reputation Engine for PeerTrust 175
5.19 Reputation Engine for REGRET 177

Acronyms

API Application Programming Interface . . . 113

ASP Answer Set Programming . . . 68

CIA Confidentiality, Integrity, and Availability . . . 202

CSA Cloud Security Alliance . . . 53

CLS Controllable Local System . . . 93

EHR Electronic Health Record . . . 59

EMF Eclipse Modelling Framework . . . 150

JDBC Java Database Connectivity . . . 125

JMS Java Message Service . . . 128

JNDI Java Naming and Directory Interface . . . 125

ICT Information and Communication Technology . . . 143

NESSoS Network of Excellence on Engineering Secure Future Internet Software Services and Systems . . . 59

OCL Object Constraint Language . . . 88

OODBMS Object-Oriented Database Management System . . . 127

P2P Peer-to-Peer . . . 5

QoS Quality of Service . . . 29

RDBMS Relational Database Management System . . . 119

SDLC System Development Life Cycle . . . 2

SLA Service Level Agreement . . . 50

SSE Secure System Engineering . . . 1

SQL Structured Query Language 119

UML Unified Modeling Language 12

UML4PF UML Profile for Problem Frames 85

WSDL Web Service Description Language 131

XML eXtensible Markup Language 23

Chapter 1

Introduction

The aim of this chapter is to set the context of this thesis and to introduce its motivation, goals and contributions. This thesis is framed along the intersection of two important research areas: Secure System Engineering (SSE) and trust management. Therefore, in Section 1.1 we provide some background knowledge on these areas, stressing the difference between trust and other related concepts like security, risk and trustworthiness. After setting the context of the thesis, we present the goals that it pursuits in Section 1.2, which also summarizes the main contributions and explains how they fit in the context of the thesis. Finally, Section 1.3 provides the list of publications that underpins the thesis, and the funding that allowed its realization.

1.1 Research Scope

This section introduces the scope of the thesis, which moves along the intersection of two research areas: SSE and trust management. The former is introduced in Section 1.1.1 and concerns with building secure systems from the ground-up. The latter, presented in Section 1.1.2, aims to translate the concept of trust into the computing domain.

The concept of trust is closely related to and has been traditionally mixed up with other concepts such as security, risk and trustworthiness. Therefore, Section 1.1.3 clarifies the intuitive relationships between these concepts, stressing their differences.

1.1.1 Secure System Engineering

The domain under which this thesis is framed is Secure System Engineering, which is an evolving discipline that unifies two important areas: system engineering and security (131) (4). The discipline takes a holistic view of security across all phases of the System Development Life Cycle (SDLC), from the initial planning and specification of security requirements to the runtime verification of security properties, as depicted in Figure 1.1. The growing importance of this discipline is reflected by industry initiatives like the Microsoft Security Development Lifecycle[1], by the creation of working groups in the area[2], and by EU-funded projects[3].

Figure 1.1: Secure System Development Life Cycle

The discipline takes a proactive side in securing systems (and in turn, the services that constitute them) as opposed to the traditional reactive approach that has been the standard in the security field (42)(131). This reactive approach has been proven to be problematic because security breaches are tackled once the harm is already done, with the subsequent loss of large amount of money and reputation (130). On the contrary, SSE proposes building the system secure from the ground-up, limiting the attack surface of the system from the very beginning, and ensuring that the enforcement of security mechanisms matches their initial specifications (35).

[1] http://www.microsoft.com/en-us/sdl/
[2] http://www.ifip.org/bulletin/bulltcs/tc11_aim.htm
[3] http://www.nessos-project.eu

The first phase of the SDLC, namely planning and visualization, studies the feasibility of the system and makes some early decisions in relation to its development. During security analysis, threats to the system, as well as security goals and concerns for the different stakeholders, are elicited. Next, design artifacts and an architecture are built in order to provide the security services that prevent the threats and that cover the security needs identified in the previous stage. This architecture is implemented in the next phase, where best practices in secure coding and secure execution platforms are used. In service-oriented environments, where systems are built by composing services, there is the need to consider secure composition platforms and approaches. The last phase of the cycle includes the runtime monitoring of the system in order to verify the enforcement of the security requirements and policies defined in previous phases. Transversally to the cycle and controlling the flow of the different activities, risk management and assurance operations are executed. The former identifies and quantifies risks that may endanger the successful achievement of the system, whereas the latter is concerned with guaranteeing that the security goals and requirements are preserved all along the cycle[4].

As further discussed in Section 1.2, this thesis focuses on engineering trust in several phases of this SDLC. Therefore, the next section provides some background knowledge on the area of trust management.

1.1.2 Trust Management

The term trust management dates back to the mid 90s and was coined by Blaze, Feigenbaum and Lacy (18) to denote new mechanisms for decoupling the identity from the authorization problem. In particular, these mechanisms simplify the two-step access control process, namely an authentication step and an authorization step, into a one-step trust decision.

Another approach to introducing trust in the computing domain was formerly proposed by Marsh (75). In this approach, namely computational trust, the idea is to study how trust is characterized in other disciplines, like psychology, game theory or economics, and replicate this trust knowledge in the computing domain for collaborative scenarios. The main difference with the traditional view of trust management is that

[4]There are different life cycles models, and some of them include phases like maintenance. Nonetheless, we abstract from concrete models and represent the crucial phases that most of them share.

trust is not longer a binary value (i.e. I trust or I do not trust), but it consists of a continuous or discrete spectrum of values that represents more accurately the level of trust in an entity.

Both approaches, namely trust management and computational trust, are encompassed nowadays under the common term trust management. The way that trust is replicated in the computing domain is through the so-called trust models. Depending on the context and the goal of the model, the tasks that it must perform may be different.

In the traditional trust management field, the model is responsible for verifying credentials and policies and for granting access to resources if the requester or provider is trusted. The model may support a negotiation protocol between the two entities in order to reveal gradually the credentials and policies, thus meeting privacy concerns.

In the branch of computational trust, the first task of the model is initializing a trust relationship between entities. Initially, as there is no local information about a partner's behaviour, external reputation information may be more weighted to make a trust decision. In addition, several types of authentication and credential systems may assist in determining an initial level of trust. Other trust models rely on the system's general tendency to trust, or *trust propensity*, defining a default trust value for newcomers considering how, in general, the entities behave in the system.

The next task of this type of model is observation and assessment, that is, monitoring the behaviour of entities and assigning trust values that depend on this behaviour. The observation can be done as an active participant of the collaboration or as a silent third party. Research in intrusion detection can be useful for this purpose (113). Suspicious activity could be misbehaviour such as breaking a system policy, or simply a high deviation from usual actions performed by the observed entity. Detection of these misbehaviours could be achieved by means of specification-based anomaly detection, in which the normal behaviour is specified using some formalization technique or language. When an observation system has detected suspicious activity (or if it has just been witness of normal behaviour), a decision must be made on what to do, leading to the update of the trust or reputation value. Trust and reputation metrics are used in order to update trust and reputation values, respectively. They consist of a set of factors, that is, the variables that are used by the metrics, and of an engine, which determines how the factors are combined to yield a final value.

As a conclusion, there are several dimensions to describe trust in the computing domain through trust models (5), which are summarized next:

- Target: entities under trust evaluation are different, ranging from users in access control systems to peers in Peer-to-Peer (P2P) networks.
- Representation: trust can be encoded by means of credentials (e.g. digital signatures), records of past interactions or ontologies in the semantic web.
- Method: hard-security approaches use authentication to decide complete trust in a user. Entities opinions, i.e. reputation, and histories of past interactions can be used to determine trust. Exchange of credentials are also used to establish trust before engaging in an Internet transaction.
- Management: the entities that determine trust can vary, from single trusted third parties to individual peers able to reason about trust in others.
- Computation: trust may be computed in many ways, using discrete values or a continuous numerical range, probabilities or confidence intervals, and considering static or dynamic time. Moreover, different algorithms exist to transfer trust values among entities.
- Purpose: trust can be used for multiple purposes, including protecting data, finding accurate information, selecting the most appropriate service, or deciding whether to provide access to resources. In the very end, the purpose of trust is supporting decision-making processes.

1.1.3 Trust, Security, Risk and Trustworthiness

Trust is an unsteady concept, because we intuitively know what it is but it is hard to put its definition down in words. Chapter 2 performs a throughout analysis of the definition of trust and all its related concepts. In this section though, we are interested in exploring the intuitive relationship between trust and other properties that are closely related to it.

As discussed in the previous section, trust in the computing domain has its origin in computer security, and in particular, in the authentication and authorization domains (43). Resources owners implicitly trust users if the latter prove their identity and this

identity is associated to access rights for these resources. As environments are more open and distributed, the reliance on identity as a trust validator becomes questionable (18). As we can intuitively guess, the ultimate goal of trust is about making a decision, for example, whether granting access to resources or not. In particular, trust is recognized to be a useful tool to empower decision-making processes under the presence of uncertainty (75).

It is again intuitive that the deployment of traditional security mechanisms on a system can help building trust in such system. If users know that a system is secure, they will most likely prefer it over another system that performs the same work and it is not secure. However, how can users know that a system is secure? There are two possibilities: either they have previous experience with the system, or a reputable entity (e.g. certification authority or security expert) tells them to believe that it is secure. In any case, it is important to notice the following subtlety: users do not trust a system because it is secure, but because they know (or they think they know) that it is secure. This is a consequence of trust being subjective.

Security is often considered one dimension or factor of trust (138). This means that even if users know that a system is secure, they still may not trust it if the system fails in other categories, such as usability, reliability, robustness or competence. If a system does not perform as users expect, or if users do not feel attracted to its interfaces, users may not trust the system (124). Therefore, security may not be enough in order to build trust. This is back up by the observation that even when companies are investing increasingly more in security (101), users do not trust them to keep their personal information safe online[5].

We can also state that under certain circumstances, security may not even be necessary at all in order to achieve trust. In particular, if users do not worry about security and they have favorable previous experiences with a system, they gain trust in the system even if the system does not implement any security measures. It is not a lack of security that may lead users not to trust, but the security incidents of past experiences or the fear that such incidents eventually occur. However, the premise that users do not worry about security is unrealistic nowadays, because the growth of incidents is

[5] http://www.itproportal.com/2013/09/14/a-question-of-trust-vs-security-do-people-actually-trust-companies-they-do-business-with-online/

raising concerns and users are more aware and fearful about security, especially after Snowden's declarations (73).

As a conclusion, we can summarize the relationship between trust and security in the following statements:

- The presence of security itself does not imply trust; however, the fact that users think that the system is secure will, in most cases, increase their trust in the system.
- The presence of trust does not imply security; users could trust the system due to other properties.
- Security is an objective property, whereas trust is a subjective property. This means that two systems can be the same secure and still being trusted differently by a group of users, due to other factors beyond security, including personal preferences.

In this thesis, we are concerned about engineering trust into software services through trust and reputation models. For us, users are in many cases services, and the system is another service or a group of services. Given that in most cases there is a positive correlation between security and trust, we discuss security in several parts of the thesis, especially when discussing factors that must be considered in order to evaluate trust. However, even when we do not explicitly mention it, we assume that there are basic security services underpinning the trust and reputation models.

Regarding risk, it is agreed that it has a strong relationship with trust (71)(57). The first point in common is that both are tools that empower decision making. They also deal with uncertainty; in the case of risk, uncertainty is represented by the probability that something unpleasant happens, whereas the way of representing uncertainty in trust depends on the model, and can range from probabilities to confidence intervals.

The relationship between risk and trust can be seen in either direction: risk as a factor of trust, or trust as a factor of risk. The former indicates that risk is a factor that can be taken into account prior to assessing trust. However, we should ask ourselves to which extent and how each risk component, probability and impact, should be weighted. The other direction of the relationships indicates that trust information can be a relevant

factor when making risk assessments, where the level of trust represents the probability used when measuring the opportunity level (71).

It seems intuitive that trust and risk have a positive correlation: the trust that an entity places in another entity to perform an action must be high if the risk involved in that action is high too; or put in other words, if there is no risk involved in an action, no trust is required. High risk implies high values at stake or high chance for deception, increasing the necessary trust to engage in the interaction. The absence of risk may imply low values at stake, minimizing (even removing) the need for making a trust decision. In this direction, Mayer, Davis and Schoorman (78) state that trust is the willingness to take risk, and the level of trust is an indication of the amount of risk that one is willing to take.

Trust and trustworthiness are two related concepts that are often used interchangeably and must be distinguished (127). Avižienis et al. (8) define trustworthiness as the "assurance that a system will perform as expected". Note that trustworthiness is a property of the system, just like security, and therefore, it is not susceptible to subjective judgement. The ideal situation is present when the trustor's trust in a trustee matches the trustworthiness of the latter (95). In that case, we say that trust is well-founded; otherwise, it is ill-founded (71). As for the relationship with risk, we can consider that the probability that measures the uncertainty partly represents the trustworthiness of the trustee (57).

In this thesis, we focus on trust and take the vision that risk is embedded in both the trust level and trust threshold of the model, following the positive correlation between trust and risk. Therefore, setting a high threshold (value over which a trust decision is positive) implies that trust must be high, and therefore, that the risk is also high. Conversely, setting a low threshold implies that risk is low, therefore trust can be low in order to proceed with the interaction. We do not consider trustworthiness.

1.2 Goals and Organization

The integration of trust into different phases of the SDLC can bring benefits to systems and, consequently, can have a positive effect on the users of such systems, who will feel more confident and willing to use them. However, there are two main stumbling blocks that prevent trust engineering from becoming a reality.

First, there is not a good understanding of trust in the computing domain. There exists a large amount of authors proposing different trust and reputation models, but few deal with systematizing the concepts behind trust and abstracting away from particular assumptions or contexts where the models are applied. To add insult to injury, trust and reputation are often mixed up, leading to still greater confusion. Therefore, one goal of this thesis is *to study the concept of trust and to build a conceptual framework that studies trust-related concepts and the relationships between these concepts.*

The second stumbling block is that there are barely tools, guidelines or approaches that provide engineers and developers with the adequate know-how on engineering trust along the SDLC. The current standard is to build trust and reputation models on top of existing systems in an ad-hoc way in order to match their specific needs, limiting the models re-usability on different systems and contexts. An approach typically used in social applications and web markets is to hard-code the reputation process in the application itself, which as stated by Farmer and Glass (34), might lead to poor, unmanageable solutions. Fixing bugs or mitigating abuse become impossible unless reputation remains an isolated module. We believe that these approaches are not adequate, and that a holistic approach where trust and reputation requirements are considered from the very beginning is required. Therefore, the other goal of this thesis is *to provide systems engineers and developers with methods and tools to manage trust and reputation in different phases of the SDLC.* Accomplishing this integration requires understanding the concept of trust, and including trust reasoning in key activities of the different phases of the life cycle.

The contributions of this thesis are summarized in Section 1.2.1, whereas its general organization is discussed in Section 1.2.2.

1.2.1 Thesis Contributions

The contributions of this thesis are the following:

- A comprehensive literature review on the integration of trust in different phases of the SDLC.
- Highlight of the constituent components of trust by the systematic analysis of multiple definitions of trust.

- A conceptual framework that analyses trust-related concepts and the relationships among these concepts, which in turn provides a common basis that allows comparing a wide range of trust and reputation models.
- A methodology to incorporate trust reasoning during Cloud sourcing decisions in the early phases of the SDLC.
- A methodology and tool that support the identification of threats in systems and organizations through the analysis of trust relationships.
- A methodology and notation that allows the elicitation of trust and reputation requirements and their integration with other functional and non-functional (including security) requirements.
- A notation that allows the specification of trust and reputation models into the system.
- A framework that allows the implementation of a wide range of trust and reputation models.
- A framework that allows building systems that evolve at runtime according to trust and reputation values.

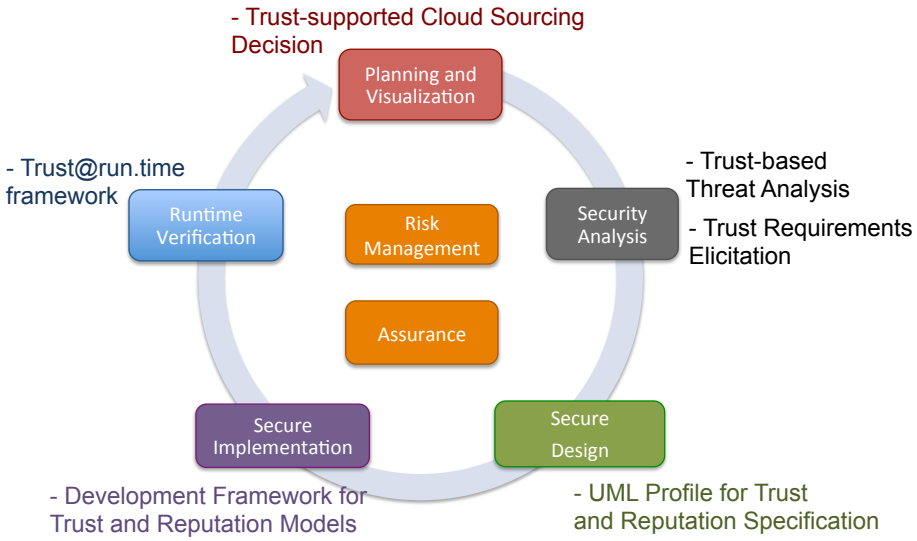
Figure 1.2 shows how these contributions are aligned with the different phases of the SDLC.

1.2.2 Thesis Outline

In this first chapter, we have contextualized this thesis in the scope of trust engineering, and we have summarized its contributions in relation to this scope.

In Chapter 2, we first provide a comprehensive literature review that analyses trust attending to different points of view. This analysis provides the required background to elaborate a conceptual framework that conveys trust-related concepts and their relationships. This framework serves as a basis for a systematic comparison of different trust and reputation models, decoupling them from particular assumptions and low-level details. The conceptual framework gathers three sets of concepts. The first set corresponds to concepts that are common to all the trust models in the literature, and

Figure 1.2: Main Contributions in relation to the SDLC



which include the factors that influence trust, the roles that entities may play, and the purpose of trust, among others. The second and third sets of concepts refer to concepts related to two classes of trust models that we consider, namely decision and evaluation models, respectively. The former revolve around the concepts of policies and credentials, whereas the latter put the stress on the assessment of trust, including trust metrics, trust factors and the sources of information. The conceptual model is used to compare a wide range of well-known trust and reputation models.

The knowledge elicited by the conceptual model is put into practice in the subsequent chapters, where we discuss methodologies, notations and tools to integrate trust in different phases of the SDLC. Thus, in Chapter 3, we focus on the early phases of the cycle, spanning the planning, analysis and design phases. As for the former, we tackle an important activity that is attracting the attention during the last years: the outsourcing of the system, or part of it, to a cloud provider. We present a methodology that supports the trust evaluation of different cloud providers according to several dimensions. This information proves to be useful when there is lack of complete information (i.e. in the presence of uncertainty) in order to minimize the probabilities of problems derived from a wrong decision, such as security or privacy incidents. The methodology requires a phase of knowledge gathering from different sources about the cloud provider. This knowledge is represented by confidence intervals, which allows representing uncertainty

explicitly. We also define an operator in order to aggregate intervals while maintaining uncertainty in each operation. The methodology output consists of a set of confidence intervals that reveals to what extent the trust in the provider differs from the initial expectations. The methodology is applied to four well-known cloud vendors under an eHealth case study.

We also integrate trust in two activities that are typically performed during the security analysis phase. The first one involves the analysis and identification of insider threats in socio-technical systems. In such systems, the organization is a key player and as such, its components and relationships must be taken into account. Therefore, by detecting implicitly assumed and misplaced trust relationships, we can infer potential threats on resources in terms of confidentiality, integrity and availability. The second contribution in this phase corresponds to an extension over the problem frames notation in which trust and reputation concepts are integrated. Therefore, by using this extension we are capable of eliciting trust and reputation requirements, integrating them with other functional and non-functional requirements.

After the requirements analysis, we move to the design phase, where an initial specification of the system with trust and reputation considerations is proposed. In order to accomplish this task, a Unified Modeling Language (UML) profile that extends several diagrams is presented. The concepts included in the profile allow designers to specify trust and reputation solutions that can be easily implemented in later stages. Not only do the profile support the description of how trust and reputation is computed, but it also enables reasoning about the purpose of trust, and in particular, for which use cases trust, reputation or both are required and why. The extended diagrams include use case, class and deployment ones. Even when sequence diagrams are not extended, we encourage their use in order to represent the interaction patterns between the system and the trust models. The profile is validated with an eHealth scenario, where physicians monitor patients through wearables that send vital signs information to the hospital servers. In this setting, it becomes very important to consider trust requirements that encompass trust relationships between patients, physicians and the wearables, as well as reputation information about the physicians.

Despite the vast amount of trust models proposed in the literature, few effort has been made on smoothing their implementation in more general contexts. Developers feel unarmed when it comes to implement these models because there are no tools or clear

guidelines on how to integrate trust models in their own systems. Therefore, Chapter 4 describes a framework that developers can use in order to implement a wide range of trust and reputation models. The requirements of the framework and its high-level architecture are discussed first, and as it is the case with the aforementioned contributions, they build upon the concepts identified in Chapter 2. Then, a low-level architecture and guidelines towards the implementation of its different aspects are introduced. The framework is designed to act as a middle-tier server that mediates between the client application and the database tiers, where the application can request trust information or send updates of trust or reputation values. The framework empowers developers to define their own metrics and the events in the application that trigger trust and reputation updates. The validation is done in the context of a social market for cloud providers, in which they can publish web services and consume services from other providers.

The use of trust and reputation in order to evolve the system at runtime is discussed in Chapter 5. In particular, we describe a development framework for trust and reputation models that is integrated in a models@run.time platform. Models@run.time is a model-driven approach that allows reasoning about a running system by means of abstractions. This approach is gaining traction among the model-driven community because it helps tackling the complexity of distributed systems and due to its self-adaptability capabilities: users can use the platform in order to reason about the state of the system and can easily determine whether a reconfiguration is required given some changing conditions. These platforms however do not usually consider security aspects, which may be a stumbling block for its widespread adoption. Therefore, the framework that we propose enables developers to include trust and reputation reasoning in the system components, which yields trust-aware systems that self-adapt at runtime based on trust and reputation values. We validate our framework in a distributed chat application by implementing several well-known trust and reputation models. We also prove that our framework entails negligible computational overhead and entails minimal effort for developers.

Chapter 6 finalizes the thesis by presenting the conclusions as well as some open research problems that require further attention from the research community.

1.3 Publications and Funding

The contributions of this thesis have been presented in various journals and international conferences. Next, we provide a list of the contributions organized by the type of publication:

Journal article ISI-JCR

- F. Moyano, C. Fernandez-Gago, and J. Lopez. A Framework for Enabling Trust Requirements in Social Cloud Applications. In *Requirements Engineering*, vol. 18, issue 4, Springer London, pp. 321-341, Nov 2013. ISI JCR Impact Factor: 1.675

International Conferences

- F. Moyano, K. Beckers, and C. Fernandez-Gago. Trust-Aware Decision-Making Methodology for Cloud Sourcing. In *26th International Conference on Advanced Information Systems Engineering (CAiSE 2014)*, M. Jarke, et al. Eds., LNCS 8484, Springer, pp. 136-149, Jun, 2014

- F. Moyano, C. Fernandez-Gago, and J. Lopez. Building Trust and Reputation In: A Development Framework for Trust Models Implementation. In *8th International Workshop on Security and Trust Management (STM 2012)*, A. Jøsang, P. Samarati, and M. Petrocchi Eds., LNCS 7783, Springer, pp. 113-128, 2013

- F. Moyano, B. Baudry, and J. Lopez. Towards Trust-Aware and Self-Adaptive Systems. In *7th IFIP WG 11.11 International Conference on Trust Management (IFIPTM 2013)*, C. Fernandez-Gago, I. Agudo, F. Martinelli, and S. Pearson Eds., AICT 401, Springer, pp. 255-262, Jun 2013

- F. Moyano, C. Fernandez-Gago, and J. Lopez. A Conceptual Framework for Trust Models. In *9th International Conference on Trust, Privacy & Security in Digital Business (TrustBus 2012)*, S. Fischer-Hübner, S. Katsikas, and G. Quirchmayr Eds., LNCS 7449, Springer Verlag, pp. 93-104, Sep 2012

- F. Moyano, C. Fernandez-Gago, and J. Lopez. Towards Engineering Trust-aware Future Internet Systems. In *3rd International Workshop on Information Systems*

Security Engineering (WISSE 2013), X. Franch, and P. Soffer Eds., LNBIP 148, Springer-Verlag, pp. 490-501, Jun 2013

- F. Moyano, C. Fernandez-Gago, K. Beckers, and M. Heisel. Enhancing Problem Frames with Trust and Reputation for Analyzing Smart Grid Security Requirements. In *Second International Workshop on Smart Grid Security*, J. Cuellar Ed., LNCS 8448, Springer, pp. 166-180, Aug, 2014

Book Chapter

- F. Moyano, C. Fernandez-Gago, B. Baudry, and J. Lopez. Engineering Trust-Awareness and Self-adaptability in Services and Systems. In *Engineering Secure Future Internet Services and Systems*, vol. LNCS 8431, no. 8431, Springer, pp. 180-209, 03/2014

Additionally to these main contributions, there are other works that have been carried out in parallel and which are listed next sorted by date:

- K. Beckers, M. Heisel, F. Moyano and C. Fernandez-Gago, Engineering Trust- and Reputation-based Security Controls for Future Internet Systems. In *The 30th ACM/SIGAPP Symposium On Applied Computing (SAC 2015)*, In Press

- F. Paci, C. Fernandez-Gago, and F. Moyano. Detecting Insider Threats: a Trust-Aware Framework. In *8th International Conference on Availability, Reliability and Security*, IEEE, pp. 121-130, Nov 2013

- F. Moyano, C. Fernandez-Gago, and J. Lopez. A Trust and Reputation Framework. In *Doctoral Symposium of the International Symposium on Engineering Secure Software and Systems (ESSoS-DS 2013)*, M. Heisel, and E. Marchetti Eds., CEUR-WS 965, CEUR-WS, pp. 7-12, 2013

- F. Moyano, C. Fernandez-Gago, and J. Lopez. Service-Oriented Trust and Reputation Architecture. In *Proceedings of the Doctoral Symposium of the International Symposium on Engineering Secure Software and Systems (ESSoS-DS 2012)*, J. Cuellar, and N. Koch Eds., CEUR-WS 834, CEUR-WS, pp. 41-46, 2012

This dissertation has been possible thanks to the support and funding of the Spanish Ministry of Education through the National Programme for Training Human Resources under the FPU (Training of University Lecturers) fellowship. Furthermore, some parts of this work have been supported by and applied to different National and European projects: NESSoS (FP7 256980), ARES (CSD2007-00004), SPRINT (TIN2009-09237), FISICCO (P11-TIC-07223), and PISCIS (TIC-6334).

Chapter 2

Understanding Trust: A Systematic Analysis

The aim of this chapter is setting up some foundations onto which we can build the rest of the chapters of this thesis. Prior to integrating trust and reputation in the different phases of the SDLC, it is necessary to systematize the knowledge around these notions. Gaining insight on trust and trust-related concepts demands a wide study of the concept of trust that spans different angles, which include revising the origins of trust management and computational trust, reviewing existing taxonomies and trust and reputation surveys, and analyzing how the concept of trust has being approximated to each phase of the SDLC.

Once we gather this knowledge, we proceed by creating a conceptual model where the most relevant trust-related concepts are stressed and related among them. This model serves us as a framework to compare different trust models under the same lens, and at the same time, it provides the basis and the core knowledge that can be integrated in different phases of the SDLC, from the early phases (Chapter 3) to runtime (Chapter 5).

The chapter consists of two main sections. Section 2.1 reviews existing works that consider trust in different contexts and phases, whereas Section 2.2 presents the conceptual model and applies it in order to compare a wide variety of trust and reputation models.

2.1 Literature Review

This section summarizes existing works on trust from three different perspectives:

- Taxonomy perspective: these works provide a classification or taxonomy of trust and reputation concepts and models.
- Computing domain perspective: here we tackle the origins of trust management and computational trust through a revision of the two main classes of trust models.
- Development life cycle perspective: we deal with works that aim to integrate trust in different phases of the SDLC.

Each of the following sub-sections deal with these perspectives in further detail.

2.1.1 Trust Taxonomies

Grandison and Sloman (43) provide a classification of trust based on its purpose. The first purpose is trusting a trustee to access the resources of a trustor. The second purpose is trusting a trustee to provide a resource to a trustor. Certification of trustees is the next purpose; in this case, trustee's identity or capabilities are certified by a third party. The next purpose is delegation, in which a trustor trusts a trustee to make decisions on its behalf. Finally, the authors discuss infrastructure trust, where trustors trust themselves (implicit trust) and the infrastructure that they use, including firmware, operating systems, local network and servers.

Trust is relevant in multi-agent systems, since agents must trust other agents in order to engage in collaborations. In this context, Ramchurn, Huynh and Jennings (104), and Sabater and Sierra (115) have provided their own conceptualisation of trust models. The former categorize trust in individual-level trust, which refers to beliefs on the honesty of other agents, and system-level trust, which involves trust as a consequence of adjusting to the rules of encounter or protocols. The latter proposes several classification dimensions for trust and reputation models. The first dimension is the conceptual model, which can be cognitive (trust as a set of beliefs) or game-theoretical (trust as an outcome of an interaction). The second dimension comprises the information sources used to gather knowledge and to make a trust decision; these sources include direct experience, witness information, sociological information and prejudice. The next

dimension is visibility types, which refers to whether trust is seen as a global property shared by all observers or is observed as a subjective property assessed particularly by each individual. Another criteria is the granularity of the model, which determines whether the model serves for a single context or allows multiple context at a time. The assumptions about agent behaviour is the next criteria; some models assume that agents may behave badly, whereas others assume that cheating behaviour will not happen. The type of information received from witness is another criteria, and the authors consider those models that use boolean information and continuous measures. Finally, the authors consider whether the model uses a reliability measure of the trust value.

In relation to multi-agent systems, Pinyol and Sabater-Mir (98) present a survey on classification dimensions by other authors under this context, and they provide their own classification dimensions. The first one is the trust dimension, where they distinguish between trust and reputation model. Essentially, the authors state that trust implies a decision, and only when the decision-making process is part of the model (e.g. the model provides a threshold computation), the model is actually a trust model. The second one is the cognitive dimension, and it refers to whether the model explicitly model the reasons behind a trust disposition, that is, if it is possible to reason about a trust decision in terms of beliefs. The next dimension is procedural, and evaluates whether the model explains the bootstrapping phase. Finally, the generality dimension states whether a model is designed for a very particular scenario or can be adapted to a variety of scenarios.

The life cycle of a trust management system is discussed by Ruohomaa and Kutvonen (113). According to the authors, this life cycle starts with the initialization of a trust relationship, where an entity uses a discovery service to find a partner. The second phase is observation, where an the behaviour and interactions of an entity are observed. In this context, intrusion detection and prevention systems may be used as a source of information for updating trust and reputation values. The next phase consists of evolving trust and reputation. As part of this phase, it is necessary to translate experiences into updates in reputation. The same authors also compare several reputation systems under a credibility taxonomy (112). They advocate that in open reputation system environments, different types of misbehaviour can occur, making it important to separate accurate from inaccurate information. This credibility taxonomy comprises three criteria: the creation and content of a recommendation, the selection

and use of recommenders and the interpretation and reasoning applied to the gathered information.

An explicit classification of trust models is performed by Artz and Gil (5). The authors divide trust models into policy-based trust, reputation-based trust and general models of trust. Policy-based trust refers to trust models in which trust is established by credentials exchange. Reputation-based trust comprises trust models that use personal experiences or others' experience to make a trust decision. General models of trust provide a broader view on trust, taking game-theoretic and psychological approaches into consideration. The relationship between trust and reputation is further discussed by Jøsang, Ismail and Boyd (56), and the authors provide a classification of reputation models centred around the semantics of ratings, the architecture (centralized or distributed) and the computation engine they use. Hendrikx, Bubendorfer and Chard (50) presents a conceptual model for reputation, which they call a reference model, where they depict relevant concepts and relationships among them. They also provide a comprehensive taxonomy of reputation systems, consisting of fourteen dimensions that encompass most features of current commercial and research models. However, they do not discuss trust concepts or relationships.

Yan and Holtmanns (138) consider that trustors' and trustees' properties are fundamental for understanding which factors affect trust. In particular, they remark that trustors and trustees possess objective and subjective properties that influence a trust relationship. The same authors then provide a taxonomy of trust models, which are categorized according to the following criteria: modelling method, that is, linguistic, graphic or mathematic description; single-property modelling or multi-property modelling; expression of trust, that is, whether trust is expressed with binary or numeral ratings; and dimension of trust expression, which refers to whether the trust value is a single value or a vector of values.

Another taxonomy for trust models is suggested by Noorian and Ulieru (89). The authors propose several dimensions scoped within two broad categories, namely hard features and soft features. As part of hard features, they classify the systems according to their rating approaches, witness locating approaches, reputation engine and information sources. As for the soft features, the dimensions considered are context and criteria similarity, adaptability to newcomers and scalability, reliability, including the accuracy

and honesty of witnesses, and reputation management parameters such as transitivity rate and time.

Other fields of study where trust is relevant are Service-Oriented Architectures (SOA) and social applications. In the former field, Wang and Vassileva (134) focus on trust and reputation as a means to leverage service selection in Service-Oriented Architectures (SOAs), and classify models according to three hierarchical criteria. The first criteria is whether the model is centralized or decentralized. The second classifies models in those that evaluate persons/agents, or resources like services. The last criteria considers whether reputation is derived from the opinions of the global population, or from a selected set of individuals. On the other hand, Zhang, Durresi and Barolli (142) state that trust research in social applications can be divided into four parts: forming initial trust, trust metrics, operations of propagating and aggregating trust, and trust management architecture. In the first category, they distinguish between trust formed by human cognition and by artificial intelligence. Trust metrics are divided into binary, scaled, multi-metric, probability, and according to whether their values are discrete or can be negative. Propagation and aggregation of trust, also referred by the authors as trust transitivity, is discussed in terms of its two basic operations: concatenation and aggregation. Trust management architecture can be centralized and distributed. According to the authors, centralized and distributed trust management lead to reputation systems, where trust is a consensus of all participants.

Note that even when several works have identified relevant concepts regarding trust models, they do not usually identify relationships among these concepts. Our conceptual framework is also built upon our own review of some of the most representative trust models in the literature, which are presented in the next section.

2.1.2 Trust in the Computing Domain

The notion of trust is brought to the computing domain through the so-called trust models. A trust model defines the way to specify and evaluate trust relationships among entities for calculating trust, being the technical approach to represent trust for the purpose of digital processing (140).

There is a huge amount of trust and reputation models in the literature, and each of them might lead to a different way of measuring trust in a system. The reason is a consequence of the inherent complexity of trust, for which no agreed definition

exists. However, several works have stated some properties of trust, including that it is a multidimensional, multidisciplinar, multifaceted, and subjective concept (138)(5).

The following sections discuss the two main approaches to modelling trust in the digital world. In either case, the final goal is using trust for empowering decision making, but each approach considers trust from a different viewpoint.

2.1.2.1 Trust as Authorization

The term trust management is coined by Blaze, Feigenbaum and Lacy in their seminal work (18), where they present the policy language PolicyMaker. By that time, the purpose of trust is to unify authentication and access control in distributed settings, simplifying the authorisation problem into a single step.

PolicyMaker constitutes the first approach towards using credentials that directly authorize actions instead of dividing the authorization task into authentication and access control. The system suggests using a query-based language (although it does not provide one) that allows determining whether a particular public key (entity) is permitted to perform a particular action according to a local policy. PolicyMaker accepts, as input, a set of policy statements, a collection of credentials, and a proposed trusted action. Depending on the credentials, the action is granted permission or not. Additional constraints might be imposed by means of annotations filters, leading to a process of simple negotiation instead of a simple yes/no answer. To sum up, PolicyMaker answers the following question: does the set C of credentials prove that the request r complies with the local security policy P?

The overall process would be as follows: first, an application sends a request to PolicyMaker. This request contains a (set of) public key(s), and an action to be performed on behalf of these key(s). PolicyMaker checks its list of assertions. If one of the assertions match the request, it grants permission. If the assertion requires more information, it annotates the missing information and sends it back to the application, which might fill this missing information and send it again. Otherwise, permission is denied.

PolicyMaker assertions can be written in any interpreted programming language, whereas in an effort towards standardization, KeyNote (17) proposes a standard assertion language. Another difference is that PolicyMaker delegates the verification of

signatures to external programs, but this verification is done by the trust management system itself in Keynote.

REFEREE (28) (Rule-controlled Environment for Evaluation of Rules, and Everything Else) provides an environment for evaluating compliance with policies, and unlike PolicyMaker, it also provides control of the evaluation process itself, since this might entail dangerous actions (e.g. network access). It also differs from PolicyMaker in that the latter does not permit policies to control credential fetching or signature verification, although KeyNote does. REFEREE also supports a more complex inter-assertion communication than PolicyMaker, since it allows assertion programs to call each other as subroutines and to pass different arguments to different subroutines, whereas PolicyMaker requires each assertion program to write anything it wants to communicate on a global blackboard that can be seen by all other assertions. REFEREE has three primitive data types, namely programs, statement lists, and tri-values, which can be true, false, or unknown. It also presents the notions of policy and credential. The former is a program that returns true, false or unknown, depending on whether the available statements are sufficient to infer compliance or non-compliance, or nothing. The latter is a program that given some initial statements as inputs, derives additional statements. Unlike KeyNote, REFEREE does not provide a standard language.

As an evolution of policy languages, we find trust negotiation models, where privacy and trust are balanced. The first example found in the literature is TrustBuilder (136), which focuses on disclosing credentials gradually to find a tradeoff between privacy and trust: an entity needs to provide a credential only if it is actually required by the policy of the other entity. The architecture relies on security agents that act on behalf of the entities. Important components of the architecture include the Negotiation Strategy Module, which decides the next message that should be sent given the current status of the negotiation, the Policy Compliance Checker, which determines the credential or policy to use given the other partner's disclosed credentials and policies, and the Credential Verification Module that verifies received credentials. TrustBuilder is a Java implementation that supports the use of X.509 certificates to encode attributes and eXtensible Markup Language (XML) to represent policies written using the IBM Trust Policy Language.

Other relevant policy languages include Cassandra (10), the family of *RT* languages (88), Sultan (44) and PROTUNE (19). The latter defines a metalanguage so that it can

be extended with application-specific predicates and annotations for confidential parts of policies. Most of these trust models incorporate the notion of delegation, according to which subjects may allow other subjects to make decisions on their behalf.

Whereas most previous existing approaches focused on client-server scenarios, Trust−$\mathcal{X}$ (15) is an XML-based framework for trust negotiations specifically conceived for a peer-to-peer environment. This is because both negotiating parties are equally responsible for negotiation management and can act as a requester or resource controller during different negotiations. In order to specify certificates (credentials) and policies, the authors suggest an XML-based language called $\mathcal{X}$−TNL. One of the novel ideas proposed by the approach is the use of trust tickets, issued upon the successful completion of a negotiation and which can be used to speed up subsequent negotiations for the same resource. Improving the efficiency of the negotiation and allowing multiple negotiations at a time is the goal of Orkphol and Jianli (93). For this purpose, they propose an extension over TrustBuilder2 (62) that introduces the Common Disclosure, a repository that accumulates every unique credential disclosed in every negotiation step from local and remote parties.

2.1.2.2 Trust as Computational Mapping of Human Trust

There is a corpus of research that focuses on modelling factors that have a direct influence on trust determination. The idea behind this vision of trust is to map the foundations of human trust into a computational setting. The purpose of trust becomes wider than authorisation and is not solely based on the possession of credentials any more.

Marsh proposes one of the first computational models of trust in his doctoral dissertation (75), where he integrates aspects of trust from several disciplines, including economics, psychology, philosophy, and sociology. By aggregating different factors, he obtains a scalar value for trust. Marsh also identifies time as being relevant to each of the variables used to compute trust. He states that X trusts Y if and only if '*X expects that Y will behave according to X's best interest, and will not attempt to harm X*'.

A cognitive approach to trust is first proposed by Castelfranchi and Falcone (24). The authors state that trust must be modelled as a set of beliefs that entities hold on each other, and as a set of goals that these entities are pursuing. They advocate that

trust goes beyond pure expectation, as Marsh suggests, as trust also involves a decision and act, and reliance and being willing to count on other parties.

Reputation models aim to represent the concept of reputation in a computational setting. In the context of the trust models we are discussing in this section, we assume that reputation is another factor, based on an aggregation of personal opinions, that can be modelled, quantified, and aggregated to help computing a trust score.

2.1.3 Trust in the Software Development Life Cycle

This section summarizes the most relevant contributions towards integrating trust in the SDLC activities, from early analysis to runtime.

2.1.3.1 Trust in Early Phases

There is a solid ground of works that consider hard security requirements at early stages of the SDLC. Some of these works focus on detecting possible attacks on the system. Sindre and Opdahl (119), and McDermott and Fox (79) propose using misuse cases and abuse cases, respectively. These methods aim to capture use cases that may be initiated by attackers or even stakeholders in order to harm the system. In a similar direction, Schneier (116) presents a formal and methodical way of capturing different types of attacks that can be performed in a system by means of attack trees.

In the realm of traditional hard security solutions, such as confidentiality or authorization, well-known approaches exist that bridge a gap between security requirements specification and security design. Jürjens (58) presents UMLsec, a UML profile for secure system development that allows designers to annotate diagrams with security information. On the other hand, Lodderstedt, Basin and Doser (68) present SecureUML, which uses the Object Constraint Language (OCL) to specify authorization constraints onto application models. Also, security patterns[1] systematize expert security knowledge into reusable artifacts, providing a toolset for non-expert designers that want to integrate security solutions during system design.

These works, however, do not address trust concerns or help designers during the specification of trust models. In this direction, Uddin and Zulkernine (125) present a UML profile for trust called UMLtrust. They provide extensions over some UML

[1] `http://www.securitypatterns.org`.

diagrams in order to represent trust information. We also propose a UML profile for trust, which is described in Chapter 3. However, their approach and focus are different than ours. For example, we consider reputation information and put the stress on how trust and reputation is to be updated and represented, which is laid aside by UMLtrust.

Other works aim to integrate the notion of risk into the requirement analysis stage. As an example, Lund, Solhaug and Stølen (70) present CORAS, a risk analysis methodology that analyses unwanted incidents for a defined asset model, and when the risk level of those unwanted incidents is beyond an acceptable threshold, several treatments are introduced to the system. Asnar, Giorgini and Mylopoulos (6) propose a concrete methodology, namely the Goal-Risk framework, to analyse and model security problems. It captures the stakeholders' goals, risks that might threaten the goals, and countermeasures required to mitigate the unacceptable risks.

Even when the concepts of trust and risk are related (see Section 1.1.3), they present several differences that justify a particular attention on trust. In this direction, the former approaches towards trust in the early stages of the SDLC come from policy languages for distributed trust management. Three remarkable examples are PolicyMaker (18), REFEREE (28) and SULTAN (44), which are further discussed in Section 2.1.2.1.

This is however a very limited scope of trust and does not take into account behavioural aspects, reputation and social relationships to determine trust. Some methodologies have aimed at a wider definition of trust, presenting methodologies to build secure systems by taking relationships between actors and agents into account. In this regard, some authors have based their work upon goal-driven methodologies to represent dependencies among actors and their views of the system. Some examples include Secure Tropos (83), KAOS (129) and SI* (77).

The first one is a methodology that extends the Tropos methodology in order to enable the design of secure systems. Actors in Tropos may depend on other actors in order to achieve a goal. Tropos captures the social relationships in the system by specifying the dependencies between actors using the notions of depender, dependum and dependee, and by modeling the actors and agents in the organization. SI* is similar, as it is based on analysing social dependencies among stakeholders, although it does not constitute a methodology on its own. SI* allows specifying the goals and views of all stakeholders of a system, considering relevant software artificats to these goals and

modelling stakeholder relations based upon structured goal models. In Chapter 3, we build a trust model upon SI* in order to detect insider threats.

As for KAOS, it is another goal-driven methodology, not so focused initially on security aspects. However, some posterior extensions (128) introduce the concepts of obstacle and anti-goal in order to analyse some trust concerns of a system. KAOS obstacle captures an undesired state of affairs that might harm safety goals (i.e., hazard) or threaten security goals (i.e.,threat), while KAOS anti-goal captures the intention of an attacker.

The analysis of social relationships is taken one step further in the works by Li, Liu and Bryant (65), Elahi, Yu and Zannone (32) and Liu, Yu and Mylopoulos (67), where actor dependency links are used as a way to identify and analyse system vulnerabilities.

The aforementioned contributions put forward the idea of capturing social aspects, but they usually fail in capturing and making all the trust relationships explicit, and above all, how trust and reputation can be used by the system-to-be. Pavlidis, Mouratidis and Islam (84) extend the Secure Tropos modelling language in order to include some trust-related concepts. In particular, they support the analysis of dependencies between actors and between actors and resources. By analysing the level of trust of these dependencies, they can determine whether an actor can be trusted to fulfil a dependency. Trust is also integrated in Role-Base Access Control model (25) by means of trust levels and a trust vector, where each component in the vector is a factor that influences trust, such as knowledge or experience.

2.1.3.2 Trust in Implementation

Few works provide developers with methodologies or tools with which they can implement different types of trust and reputation models. Suryanarayana, Diallo and Taylor (123) propose the *4C* framework, where they describe a trust model as a composition of four sub-models, namely the content sub-model, the communication sub-model, the computation sub-model and the counteraction sub-model. For each sub-model, they identify the main building blocks that are present in existing reputation models. In the end, using an Java-based editor and following a personalized XML schema, they create a XML document where a trust model is described according to these building blocks. Then, they can use the PACE Support Generator to create software components that integrate within the PACE architectural style, which is further described in the work

by Suryanarayana et al. (122). In this latter work, the authors study the feasibility of an event-based architectural style in order to provide architects with guidelines on how to include trust models into decentralized applications.

While the application scope of the previous framework is general, the *Pythia* (135) framework targets owners of blog sites who want to integrate reputation dynamics in the latter. This framework follows a plug-in architecture, where plug-ins provide the instantiation required for a concrete domain (i.e. a concrete web site). Therefore, a user who wants to use a *Pythia*-based system should first download the plug-in for that specific system. An important component of the framework is the rules engine, which is edited by users to decide how reputation is to be evaluated.

The accommodation and integration of different trust or reputation models is also addressed. For example, Lee and Winslett (62) propose an extensible framework that supports the adoption of different negotiation-based trust models. The framework abstracts away concrete implementation details of different negotiation models and provides generic building blocks that allow implementing new ones. In a similar direction, although more focused on tackling the context dependency of trust, Huynh (53) proposes the Personalized Trust Framework (PTF). This framework consists of a rule-based system that uses semantic technologies (e.g. ontologies) to capture expert knowledge about different contexts and to apply the most suitable trust model according to these contexts. The idea is to replicate the trust evaluation process carried out by humans in a computational setting, because humans are capable of determining the context under which a trust evaluation is to be performed.

Customizable trust models are another design and implementation aid. Examples of these are the SECURE platform (23) and SCOUT (48). They do not provide much flexibility with regards to implementing new models, because they are bound to an underlying model. However, they allow their customization by defining or changing some of the inner components. The former relies on a cost-benefit analysis represented by combined cost-PDFs (Probability Density Functions). The idea is, for every possible outcome of an action, to consider all possible costs and benefits the user might incur in. If the final combined cost-PDF shows that the benefits outweigh the other outcomes' costs, then the action proceeds. Otherwise, further interaction is arranged. The latter is made up of three services that implement the model: the evidence gathering service, the belief formation service and the emotional trust service.

Vinkovits, Reiners and Zimmermann (132) present a user-centered approach that allows non-security experts to include trust and reputation into their applications. The approach is a model-driven system that starts by allowing users to select several trust requirements. From these requirements, the system proposes different trust frameworks, which have been previously implemented by trust experts. After choosing one of these frameworks, users can fill in the remaining code in order to adapt it to their particular applications.

2.1.3.3 Trust at Runtime

There is a growing interest in how trust can assist evolving systems over their lifetime. Self-adaptive systems can take trust and reputation information in order to leverage runtime reconfiguration decisions, especially in the areas of multi-agent systems, component-based systems and service-oriented systems. In some cases, trust is considered in its hard variant, where trust is seen as an aggregation of Quality of Service (QoS) and security properties. In other works, the soft variant of trust is used (106), which means that social aspects such as reputation or preferences are taken into account.

As mentioned earlier, trust is seen as a powerful tool to leverage decision-making even with partial information. This fact is especially remarked in STRATUS (110), a set of technologies that aim at predicting and responding to complex cyber attacks. When it detects an attack, the platform switches to back-up components and finds alternative pathways of communication. The trust model that supports this platform (109) is based on conditional trust, that is, trust in certain capabilities of a system. The authors argue that experience-based trust is not useful because configurations in cyber attacks change frequently, laying statistical analysis useless. They propose ways to make the most out of the little information available, and they introduce concepts like *contagion* that allows formalizing trust in a host based on the distance from an infected host.

A classical scenario of application of trust is multi-agent systems (104), where Vu et al. (133) propose trust-based mechanisms as a way to self-organize the agents in case of deceitful information. In particular, the trust value of an agent towards another one is an aggregate of direct experiences and testimonies. The use of artificial intelligence, and concretely, machine learning together with trust in order to adapt the behaviour of agents is proposed in the work by Klejnowski, Bernard, Hähner and Müller-Schloer

(60). They propose an architecture where there is an *observer* component that gathers information about the agent and presents it to the *controller* in two views: a long-term and a short-term one. The *controller* finds a suitable behaviour according to this information. Given that new unexpected situations might arise, agents must be able to try out new strategies and learn which ones provided the best results.

Given the highly open and distributed nature of service-oriented environments, the traditional use of trust is for either protecting providers from potentially malicious clients or for shielding clients against potentially malicious providers (e.g. providers that publish a higher QoS than offered). As an example of the first situation, Conner et al. (29) present a feedback-based reputation framework to help service providers to determine trust in incoming requests from clients. As an example of the second approach, Crapanzano et al. (31) propose a hierarchical architecture for SOA where there is a so-called super node overlay that acts as a trusting authority when a service consumer looks for a service provider.

In both, component- and service-oriented systems, an important research area is determining the level of trust, or the trustworthiness, of the system as a whole, or of individual subsystems (i.e. services or components). In case that the trust value is too low, a reconfiguration takes place in order to try to improve it. In this direction, Haouas and Bourcier (47) present a runtime architecture that allows a service-oriented system to meet a dependability objective set up by an administrator. System dependability is computed by aggregating ratings provided by service consumers regarding QoS attributes. Then, a reconfiguration manager may look up other available services to meet the dependability objective. Dependability of the system is computed by the aggregation of each service dependability. In turn, each service dependability is computed by aggregating a weighted average of ratings provided by service consumers regarding QoS attributes (e.g. response time) of service providers. The reconfiguration manager is in charge of querying the service broker to find the available services that can meet the dependability objective.

Following a similar line of work in component-based systems, Yan and Prehofer (139) discuss an adaptive trust management system where several quality attributes can be used to rate the trustee's trustworthiness, such as availability, reliability, integrity or confidentiality. Assessing these attributes requires defining metrics and placing monitors

to measure their parameters. Finally, trust is assessed at runtime based on the trustor's criteria and is automatically maintained by changing among trust control modes.

These previous works are highly focused on QoS-based trust, where trust is an aggregation of dependability and security attributes. Subjective factors affecting trust and reputation concepts, with which we deal, are out of discussion. The social notion of trust is used by Psaier et al. (100) in their self-adaptation framework. In particular, their trust model uses the concept of trust mirroring and trust teleportation. The former implies that actors (i.e. services) with similar interests and skills tend to trust each other more than unknown actors, whether the latter denotes that the level of trust in a member of a group is transferable to other members of the same group. The adaptations consist of reconfiguring the network by opening channels to provide new interactions and by closing channels to hinder misbehaving nodes to further degrade the system function. The trust model is used to help choose among a set of new candidate nodes with which to communicate.

Another way to measure the (mis)behaviour of components is by comparing its interactions with the models in their contracts. In this direction, Herrmann and Krumm (51) propose security wrappers that monitor the activity of the components. Depending on the deviation of the components' behaviour with respect to their contract, a positive or negative report is issued and sent to the trust information system, which calculates a trust value for the component. In turn, this trust value is used to determine the intensity of the monitoring activity by the wrappers. This scheme was enhanced by Herrmann (52) in order to take the reputation of components' users into account so as to prevent deliberate false feedbacks. In this regard, a common problem in any setting where different entities rate each other is discerning fair from unfair ratings. Phoomvuthisarn, Liu and Zhu (97) propose a reputation mechanism for SOAs environments that allows services to retrieve other services' reputation through auctions that ensure incentives for truthful reporting.

Other works focus on the self-adaptation of trust models to match and reflect the status of the system (69), and on considering the trust in the self-adaptation process itself (40).

Finally, Kiefhaber et al. (59) present the Trust-Enabling Middleware, which provides applications running on top of it with methods to save, interpret and query trust related information. The middleware provides self-configuration and self-optimization and its

goal is balancing the workload of nodes by relocating services. The middleware also uses built-in functions to measure the reliability of nodes by considering packets losses.

2.2 Trust Conceptual Model

In this section, we introduce and discuss our trust conceptual model, which builds upon the knowledge gathered in the previous sections. The goal of this model is to abstract away the particularities of concrete trust models and to provide a consistent set of concepts and relationships among these concepts that assist the comparison of different classes of trust models, as well as the integration of trust concerns in different phases of the SDLC.

In the following sub-sections, we elaborate on the definition of trust and depict some of the most relevant concepts that are related to this notion. Then, the concept of trust model is introduced and a classification of trust models is provided. This classification is used as the basis to decompose the concepts that underlie in trust models, which we describe next. Finally, we describe how we use these concepts as a yardstick for comparing different well-known trust and reputation models.

2.2.1 Trust Definitions

Many definitions of trust have been provided over the years. This is due to the complexity of this concept, which spans across several areas such as psychology, sociology, economics, law, and more recently, computer science. The vagueness of this term is well represented by the statement "trust is less confident than know, but also more confident than hope" (82).

In this section, we revise the definitions that have been mostly considered in the literature of computational trust and reputation models. We advocate that making an effort to understand this term and its implications is crucial if we want to implement meaningful models. On the other hand, understanding trust and reputation allows for a better trust-related concepts identification as well as for building a more comprehensive conceptual framework for trust models comparison. Definitions are presented in chronological order.

Gambetta (37) defines trust as "a particular level of the subjective probability with which an agent will perform a particular action [...] in a context in which it affects our

Table 2.1: Trust Definitions

1988	1991	1995	1996	2000	2002	2005	2011
(37)	(20)	(78)	(80)	(43)	(85)	(104) (91) (113)	(142) (48)

own action". Mayer, Davis and Schoorman (78) advocates that trust is a "willingness to be vulnerable to another party". McKnight and Chervany (80) explain that trust is "the extent to which one party is willing to depend on the other party in a given situation with a feeling of relative security, even though negative consequences are possible". Mui, Mohtashemi and Halberstadt (85) define trust as "a subjective expectation an agent has about another's future behavior based on the history of their encounters". For Olmedilla et al. (91), "trust of a party A to a party B for a service X is the measurable belief of A in that B behaves dependably for a specified period within a specified context (in relation to service X)". Ruohomaa and Kutvonen (113) state that trust is "the extent to which one party is willing to participate in a given action with a given partner, considering the risks and incentives involved". Finally, Har Yew (48) defines trust as "a particular level of subjective assessment of whether a trustee will exhibit characteristics consistent with the role of the trustee, both before the trustor can monitor such characteristics (or independently of the trustor's capacity ever to be able to monitor it) and in a context in which it affects the trustor's own behavior".

Table 2.1 summarizes all the definitions used as inputs to build the concepts cloud depicted in Figure 2.1. Definitions were processed following several rules. A word that appears several times in the same definition is counted just once. We only take into consideration words that mean something by themselves and do not require surrounding words to mean something (e.g. *particular level* does not make sense separately). If two words with the same meaning appear either in plural and singular, it is expressed in singular. Dependability is split into security and reliability. Party, agent, entity, trustor and trustee are named as entity. Most words are adjectives and nouns, since they are more meaningful without a context than verbs, but some relevant verbs are considered as well. Assessment is used in place of *quantifiable*, *measurable*, *describable* and alike terms. The resulting concepts were introduced in Wordle[2].

[2] `http://www.wordle.net/` is a free online tool to generate words clouds.

Figure 2.1: Concepts Cloud for Trust Definitions

The figure reveals that entity is the core concept, which is obvious as trust makes no sense if there are neither entities that trust nor entities in which to trust. Context is another relevant concept since trust is highly context-dependent. Other important concepts entail uncertainty such as subjective, belief, willingness or expectation. They show that trust implies uncertainty about an entity's behaviour. It is also important to note that even though the concept of risk is not explicitly present in all the definitions, a careful reading reveals that it is indeed implicitly considered in almost all of them. For example, in his definition, McKnight (80) states that "... negative consequences are possible", and Mayer (78) claims that trust is willingness to be vulnerable.

As a wrap-up, the notion of trust is present when there is uncertainty and risk during the interaction of two or more entities that need to collaborate in a particular context. If the entity placing trust knows the outcome in advanced without any uncertainty, trust is not necessary. If the entity placing trust knows that there is no risk involved in the outcome of the interaction, trust is not necessary. If there is no interaction between two entities, trust may still make sense, but it is not necessary either.

After our review and given that no definition covers all the concepts that we believe are the most important, we propose the following definition: *trust is the personal, unique and temporal expectation that a trustor places on a trustee regarding the outcome of an interaction between them that affects the trustor.*

2.2.2 Trust Models: Definition and Classification

A trust model is an abstraction of the dynamics of trust and defines the way to specify and evaluate trust relationships among entities for calculating trust. Another way of

defining a trust model is as the technical approach to represent trust for the purpose of digital processing (140).

Trust models are very heterogeneous due to many factors, including the trust definition on which they are built or their application domain. In order to provide a conceptual framework for trust models we first establish a high-level classification. Note that this task is not straightforward and other classifications have been proposed (5). We advocate that the following classification covers the two main branches that gave rise to the adoption of trust in the computational world, as further described in Section 2.1.2.

- Decision Models. Trust management has its origins in these models (18). They aim to make more flexible access control decisions, simplifying the two-step authentication and authorization process into a one-step trust decision. *Policy models* and *negotiation models* fall into this category. They build on the notions of policies and credentials, restricting the access to resources by means of policies that specify which credentials are required to access these resources.

- Evaluation Models. These models have their origin in the work by Marsh (75). Their intent is to evaluate the level of trust that an entity can place on another entity by considering factors that have an influence on trust relationships. An important sub-class of the former are *propagation models*, in which existing trust relationships are exploited to generate new trust relationships. Another important sub-class are *reputation models*, where a reputation scored is derived from the aggregation of other entities opinions.

Making a classification is important as it eases the extraction of common features between different classes of models. It is not useful to compare decision models such as PolicyMaker (18) with evaluation models like eBay's reputation system (107), because their nature, workings and purposes are completely different. However, comparison makes sense within each model class, because models in the same class exhibit similar features that can be compared. The next section elaborates on the underlying concepts of trust models.

2.2.3 Trust Models Concepts

For the sake of simplicity, we divide our conceptual framework into three concepts blocks. The first block contains concepts that are applicable to any trust model, independently from its class. They are concepts tightly coupled to the notion of trust. The next two blocks gather concepts specific to the classes of models identified in Section 2.2.2. The concepts are depicted following a UML-like notation, because it is widely known language to represent concepts and relationships.

2.2.3.1 Common Features

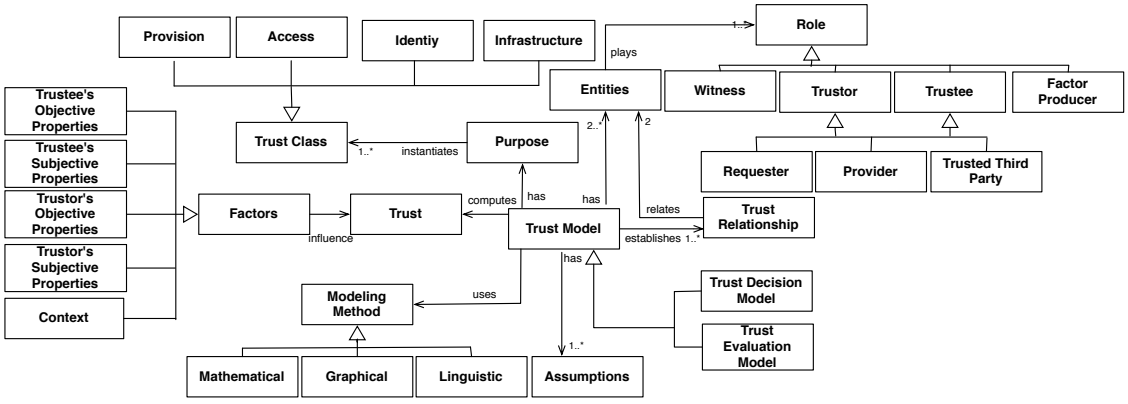
Trust is computed by a trust model that must have, at least, two entities which have to interact in some way. In any trust setting, an entity plays a role, or even several ones. In the most general case, these roles are trustor, the entity which places trust, and trustee, the entity on which trust is placed. However, depending on the context and complexity of the model, other roles are possible. For example, an entity can be a witness if it tells its opinion about an entity based on observations or its own experience. Also, an entity may be a factor producer, which means that the entity is in charge of generating a factor that has influence in the trust or reputation computation. Some specializations of trustors and trustees include a requester of a service or resource, the provider of a service or resource, or a trusted third party that issues credentials or gathers feedbacks to compute a centralized reputation score. Once we have a trustor and a trustee, we say that a trust relationship has been established.

In any trust model, establishing a trust relationship has a purpose. According to Jøsang et al. (56), a trust purpose is an instantiation of any of the following trust classes identified by Grandison and Sloman (43): access trust, provision trust, identity trust, and infrastructure trust (considering delegation a sub-class of provision trust). The instantiation is due to the fact that trust is context-dependent, one of the most important properties of trust, since it influences all the other concepts, such as the purpose, the type of entities and the role that they can play. Other factors, in addition to the context, that have an influence on trust are trustee's and trustor's subjective properties such as honesty, confidence, feelings, willingness or belief; and trustee's and trustor's objective properties like observed behaviour, security, ability, a given set of standards or reputation (138).

Note that trust can also be conceived as a strong belief about a given property of the trustee. From a theoretical perspective, there is no purpose under this trust conception. Yet we are interested in trust models from a more pragmatic perspective. Thus, trust in a given property can eventually assist in making a decision for some purpose. For instance, if an entity believes that another entity is competent to encrypt files, it will select the latter among other less qualified candidates (according to the entity's belief). In this example, the purpose will therefore be the provision of an encryption service (i.e. provision trust).

A trust model may also make some assumptions, such as "entities will provide only fair ratings" or "initial trust values are assumed to exist", and might follow different modelling methods, including mathematical, linguistic and graphical. The resulting conceptual model that gathers these concepts is depicted in Figure 2.2.

Figure 2.2: Common Concepts for Trust Models



2.2.3.2 Concepts for Trust Decision Models

Trust decision models use policies, which specify the conditions under which access to a resource is granted. Policies are written in a policy language, which might consider policy conflicts resolution. The conditions under which accesses are granted are expressed by means of credentials, signed logical statements that assert that an entity is which it claims to be, or that it is member of a group. Credentials might have different formats, including X.509 certificates[3] and XML. The component that glues credentials and poli-

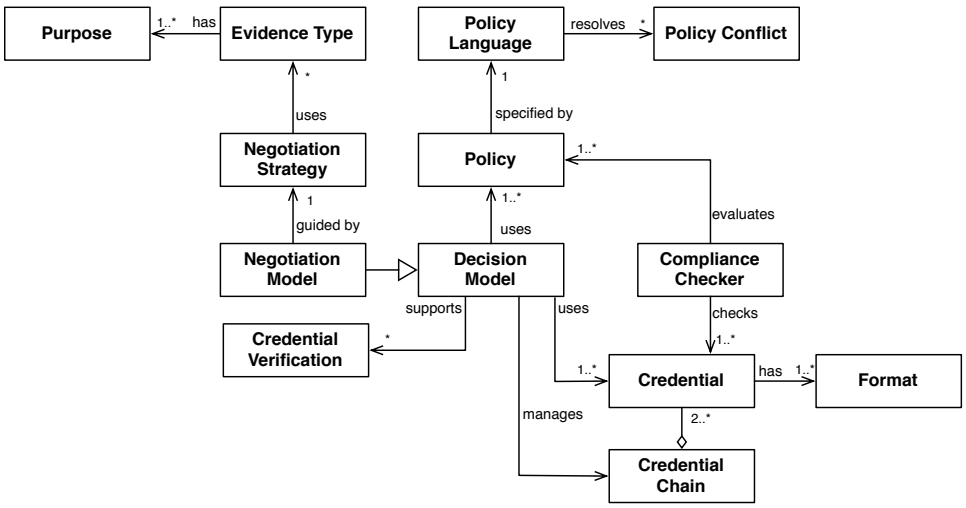
[3] https://www.ietf.org/rfc/rfc2459

cies together is the compliance checker, in charge of checking whether the credentials satisfy the policies. Likewise, the model might also support the search for a credential through credential chains, as well as the verification of the validity of credentials.

Negotiation models, a specialization of decision models, add a protocol or negotiation strategy, during which two entities perform a step-by-step, negotiation-driven exchange of credentials and policies until they decide whether to trust each other or not. Negotiation models can use evidence types, which represent information about the negotiation process (e.g. some steps of the negotiation process have been recently performed) and which have a purpose, in most cases, the optimization of the negotiation.

The conceptual model for decision models is depicted in Figure 2.3.

Figure 2.3: Concepts for Decision Models



2.2.3.3 Concepts for Trust Evaluation Models

Evaluation models often follow a life cycle with two stages. First, a bootstrapping phase might be required to assign initial trust values to the entities of the system and to every newcomer. Trust propensity is a concept related to the bootstrapping phase and it refers to the propensity of the model towards high or low trust values in the beginning. Second, a trust assessment process is performed in order to assign trust values to entities according to certain factors. This process usually involves some monitoring in order to feed these factors with accurate data.

Trust relationships are tagged with a trust value that indicates to what extent the trustor trusts the trustee. This value has a dimension, which indicates whether it is a single value or a tuple of values. Also, trust values have semantics, which can be represented by two dimensions: objectivity and scope. The former refers to whether the measure comes from an entity's subjective judgement or from assessing the trusted party against some formal criteria. The latter specifies whether the measure is done against one factor or against an average of factors.

In many cases, the model also includes the process to define a trust threshold, embedding the trust decision in the model itself. If the trust value is above the threshold, the trustor is assumed to trust the trustee and can proceed with the interaction. This is common in traditional formal trust models, such as the one by Marsh (75), where the trust decision is as important as the dynamics to update the trust values.

Trust values are assigned to relationships by a trust assessment process. The concept of trust assessment, and all the concepts related to it, are the most important ones in evaluation models, as they may become the model signature, what makes a model different from others. In order to carry out the trust assessment process, trust metrics are used. Trust metrics use factors, such as risk, utility, past experience or observed behaviour, and combine them in order to yield a final score for the measured attributes. The most general attributes that are measured are simply trust and reputation. Yet depending on the application domain, more specialized attributes can be measured, such as *reliability of the seller* in an e-Commerce scenario. Trust metrics use computation engines, which determine the way factors are combined, and which range from simple summations to more sophisticated ones like belief, Bayesian, fuzzy or flow engines. Jøsang (56) provides an overview of trust and reputation engines.

Sources of information that may feed the metrics include direct experience (either direct interaction or direct observation), sociological and psychological factors, and third party referrals. Reputation models use public trust information from other entities to yield a reputation score. Reputation models can be centralized, when there is an entity in charge of collecting and distributing reputation information; or distributed, when there is no such a role and each entity has to maintain a record of trust values for any other entities, and send out this information to the rest of entities. Regardless of which information sources are used to compute trust values, the model might consider

how certain or reliable this information is (e.g. credibility of witnesses), and might also consider the concept of time (e.g. how fresh the trust information is).

Most evaluation models follow a game-theoretic approach, where the trustor determines trust in a trustee by examining the outcomes after each interaction. Trust is usually defined in these models as a subjective expectation about the outcome of an interaction with another entity. Fewer evaluation models follow a cognitive approach, where trust is primarily determined by the mental state of the entities, which usually comprises a set of beliefs on other entities.

Propagation models, a sub-class of evaluation models, assume that several trust relationships have already been established and quantified. They aim to compute trust among entities with no direct interaction, creating new trust relationships by disseminating the trust values information to other entities. Some models assume that trust is transitive and exploit this property, although transitivity is not considered as a property that holds for trust in many cases (27). New trust values are often computed by means of operators, and in most models, we find two of them: a concatenator and an aggregator. The former is used to compute trust along a trust chain, whereas the latter aggregates trust values computed for each path into a final trust value. For example, Agudo, Fernandez-Gago and Lopez (1) use a sequential and a parallel operator in order to compute trust along a path. Subjective logic (55) uses a discounting operator to compute opinions along different trust paths, and a consensus operator to combine them into a final opinion.

All the concepts discussed are shown in Figure 2.4.

2.2.3.4 Concepts for Reputation Models

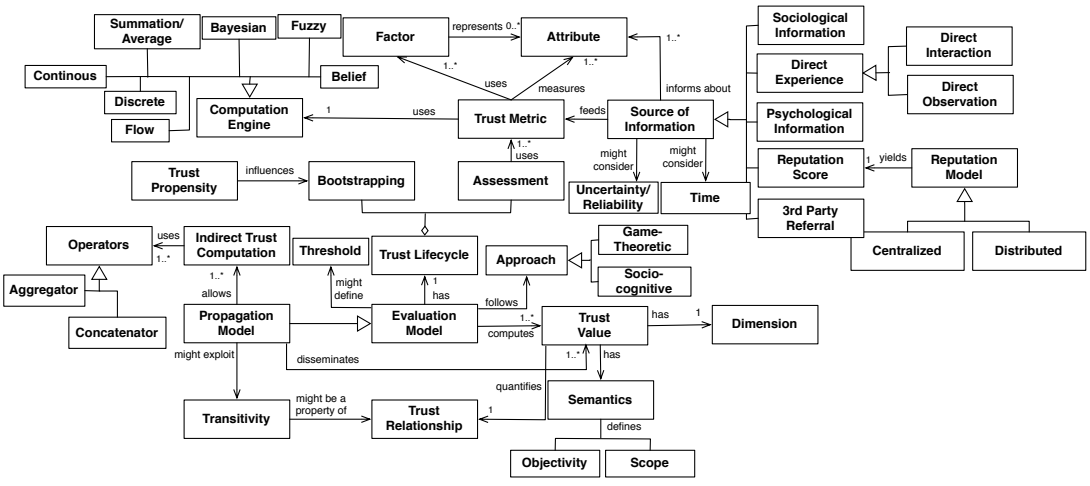
Whereas the boundaries between trust and reputation are often blurry in the literature, it is agreed that reputation is a factor that may influence trust decisions (56). We already mentioned that reputation models are a sub-class of evaluation models and that reputation models can be centralized or distributed, but it remains unclear the criteria under which we consider that a given model is a reputation model or a trust model, beyond the fact that the authors refer to it as one or another.

According to Pinyol and Sabater-Mir (98), the difference between a trust model and a reputation model is that the former embeds the trust decision itself, by means of a well-defined trust threshold. Given that few models in practice include the threshold,

we relax this condition and we consider that the difference lies in the subjective nature of trust. In particular, if given the same context and target entity, the output of the model does not depend on the entity that initiates the computation, it is a reputation model. Otherwise, if given the same context and target entity, the output of the model depends, among other factors, on the entity that initiates the computation (i.e. different entities can obtain different values for the same entity and under the same context), it is a trust model. The fact that the computation depends on the evaluating entity reveals that there are subjective factors in play, such as preferences, beliefs, etc. that are shaping a unique relationship between the evaluating entity and the evaluated entity, who become the trustor and trustee, respectively, of such relationship. This reasoning is also aligned with the intuitive understanding that reputation is more objective than trust.

We consider that reputation models have their own concepts, and we base such concepts on web reputation systems (34). The core concept is a reputation statement, which is a tuple of a source, a claim and a target. A source is any entity in the system capable of making claims about another entity of the same system, which is called a target. Reputation models use reputation engines that take reputation statements about a given target as inputs and yield a reputation score for that target.

Figure 2.4: Concepts for Evaluation Models



2.2.4 Comparison Framework: A Case Study

Not only do the concepts identified in the previous sections provide insight on computational trust, but they also allow comparing trust models under the same lens. As a way to validate our framework, we have chosen a set of relevant trust and reputation models that represent the classes discussed earlier. The models are Marsh's model (75), PolicyMaker (18), Jøsang's belief model (55), REGRET (114), TrustBuilder (136), eBay reputation model (107), Falcone et al. (33), Trust$-\mathcal{X}$ (15), PeerTrust (137) and Agudo et al. (1).

As depicted in Figure 2.5, the comparison framework allows comparing trust models at two different layers. At a higher level, the framework compares different classes of trust models with regard to the common concepts. Then, models belonging to the same class are compared according to class-specific concepts. The classes are those discussed in Section 2.2.2, and we incorporate pure propagation models because even when we consider them a special class of evaluation model, they might lack some of the evaluation models concepts.

Figure 2.5: General Framework for Trust Models Comparison

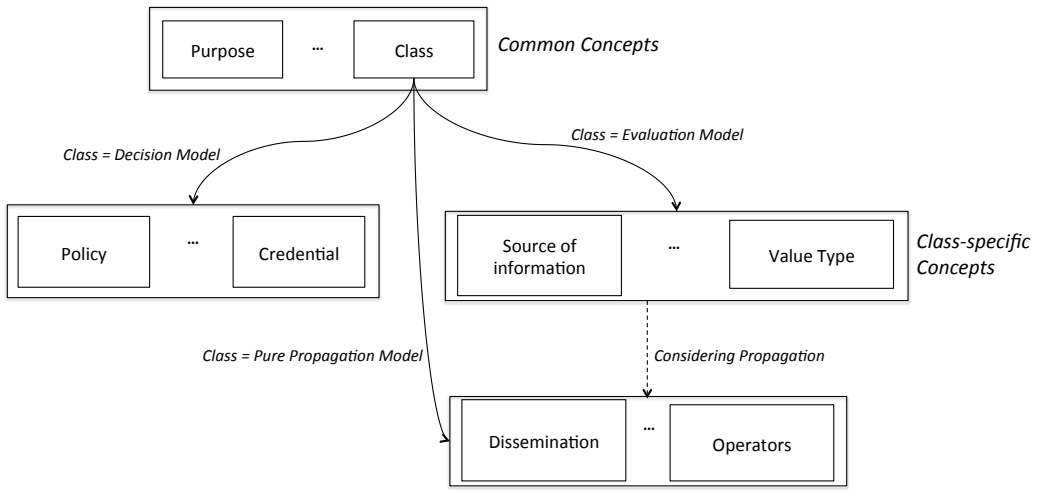


Table 2.2 shows the comparison among these models under the lens of their common features. In Table 2.3 we compare the trust decision models, whereas trust evaluation models are compared in Table 2.4 and Table 2.5. Note that the classification has been made according to the features explicitly presented by the corresponding authors, and

that due to the diversity of the models, in some circumstances the classification for some concepts is subjective according to our own interpretation.

Table 2.2: Common Features Comparison

Model	Role	Purpose	Class	Method
PolicyMaker	R/P	AT, IT	DM	Linguistic
TrustBuilder	R/P	AT, IT	DM	Linguistic
Trust−$\mathcal{X}$	R/P	AT, IT	DM	Linguistic
Marsh's	T	AT, PT	EM	Mathematical
Falcone et al.	T, W	AT, PT	EM	Graphical, Mathematical
REGRET	R/P, W	AT, PT	EM	Mathematical
PeerTrust	R/P	PT	EM	Mathematical
eBay	R/P, TTP	PT	EM	Mathematical
Jøsang's	T	AT, PT	EM	Mathematical
Agudo et al.	T	AT, PT	EM	Graphical, Mathematical

T=trustor/trustee, R/P=requester/provider, W=Witness, TTP = Trusted Third Party, AT=Access Trust, IT=Identity Trust, PT=Provision Trust, DM=Decision Model, EM=Evaluation Model

Table 2.3: Decision Models Comparison

					Trust Negotiation	
Model	P. Language	C. Format	CC	CV	Strategy	ET
PolicyMaker	PolicyMaker	PGP's sig, X.509 cert	-	-	-	-
TrustBuilder	XML, IBM's TPL	X.509 cert	✓	✓	✓	-
Trust−$\mathcal{X}$	$\mathcal{X}$-TNL	$\mathcal{X}$-TNL	✓	✓	✓	✓

CC=Credential Chaining support, CV=Credential Verification support, ET=Evidence Type, -=undefined or not explicitly mentioned

By observing Table 2.2, we observe that the purpose of decision models is often either access trust (a provider wants to protect a resource from malicious requesters) or identity trust (determine trust in a requester based on its identity), whereas the purpose of evaluation models is either protecting a requester from malicious providers (provision trust), or protecting providers from malicious requesters (access trust).

Table 2.4: Evaluation Models Comparison

				Source of Information				
Model	Approach	Dimension	C.Engine	DI	SI	PI	R	TPR
Marsh's	GT	1	Continuous	✓	-	✓	-	-
Falcone et al.	SC	1	Fuzzy	✓	✓	✓	✓	✓
REGRET	GT	1	Continuous	✓	✓	-	-	✓
PeerTrust	GT	1	Continuous	✓	-	-	D	✓
eBay	GT	1	Summation	-	-	-	C	✓
Jøsang's	-	3	Flow/Belief	-	-	-	-	✓
Agudo et al.	-	1	Flow	-	-	-	-	✓

DI=Direct Interaction, SI=Sociological Information, PI=Psychological Information, R=Reputation, TPR=Third Party Referral, C=Centralized, D=Distributed, GT=Game-Theoretic, SC=Socio-Cognitive, -=undefined or not explicitly mentioned

Table 2.5: Evaluation Models Comparison (II)

Model	Threshold	Indirect Trust Calculation	Uncertainty	Time
Marsh's	✓	✓	-	✓
Falcone et al.	-	-	✓	-
REGRET	-	-	✓	✓
PeerTrust	-	-	✓	✓
eBay	-	-	-	✓
Jøsang's	-	✓	✓	-
Agudo et al.	-	✓	-	-

-=undefined or not explicitly mentioned

We also observe that decision models follow a linguistic modelling method, embodied in the policy and credential languages. Regarding evaluation models, most use mathematical methods. Additionally, Agudo et al. use graph theory and Falcone et al. use Fuzzy Cognitive Maps, and therefore they both use graphical and mathematical modelling methods.

Regarding the roles, decision models exhibit the requester/provider pair, as they are usually intended for scenarios with Internet transactions. Evaluation models do

not usually specify concrete roles beyond trustor and trustee, except for those that include a witness that provides third-party referrals, or a trusted third party that stores reputation information.

As for decision models, we see in Table 2.3 that PolicyMaker is a pure policy model, whereas TrustBuilder and Trust$-\mathcal{X}$ include a negotiation strategy, as well as credential chaining support and credential verification. The latter also includes evidence types in order to prove that some steps of the negotiation protocol have succeeded.

Table 2.4 shows that evaluation models follow a game-theoretic approach, except for Agudo et al. and Jøsang's, which are pure propagation model, and Falcone et al., which is a socio-cognitive evaluation model. Most models provide a single-dimension value, except for Jøsang's, which provides a vector of values that represent belief, disbelief and uncertainty. Semantics have been omitted as all trust models consider trust under some sort of *subjective* judgement (and not as formal measurements) and take into account *general* properties (and not specific ones).

Regarding the sources of information, most of the models use third party referrals, except for Marsh's[4]. The most complete model in terms of sources of information is Falcone et al.'s, because it considers also psychological information in terms of beliefs. PeerTrust is a distributed reputation model, whereas eBay's is centralized. Sociological information, that is, the fact that an entity belongs to certain groups, is only used in REGRET and Falcone et al.'s. The latter also uses reputation as a factor to determine trust.

Regarding Table 2.5, Marsh's model is the only one that incorporates the computation of a trust threshold. The main difference between the two pure propagation models, namely Agudo et al.'s and Jøsang's, is that the former does not consider uncertainty (i.e. credibility), whereas the latter does. Note that the main purpose of propagation models is trust dissemination, or in other words, the calculation of indirect trust relationships. However, Marsh's model, even when it is not a propagation model itself, it provides the means to disseminate trust information.

[4]Marsh's model tackles trust dissemination, but it does not use third party referrals for computing trust. Furthermore, we consider that the initial relationships on which the pure propagation models work are a type of third party referral

Chapter 3

Incorporating Trust Engineering in Early Phases of the SDLC

This chapter addresses several key considerations when integrating trust in activities that are part of the early phases of development, from planning to design.

During the planning phase, there is an ever-increasing need to decide whether the system or a part of it should be moved to the Cloud, activity known as cloud sourcing. This provides several benefits in terms of scalability and cost-effectiveness, but it raises security concerns as to who can access the deployed data or services. Cloud sourcing has a high impact on the design of the system outsourced, given that it entails a partial loss of control and consequently, potential threats to security. The first important decision that we must make is to which cloud vendor we are moving part of the system. This first decision can be key to the rest of the design and we advocate that trust evaluation of cloud vendors can help cushion the aforementioned loss of control and potential security problems.

Along the second phase, namely security analysis, there are two important activities to cover. First, in order to achieve trust-aware solutions, we need to capture trust requirements. However, whereas the security community has traditionally focused on providing tools and mechanisms to capture and express hard security requirements (e.g. confidentiality), little attention has been paid to trust and reputation requirements. We argue that these soft security requirements can leverage security in open, distributed, heterogeneous systems and applications and that they must be included in early phases of the development process.

Another typical activity performed during this phase is threat analysis. Many threats, and particularly the ones that result in the most harmful incidents, are due to insiders. In most cases, these threats originate from false or implicit assumptions about trust relationships in the organizational context of the system. Therefore, and in order to guarantee the design of more secure systems, it is required that trust relationships are explicitly identified, analysed and quantified. From these trust relationships and modelling the criticality of the assets of the system, we can infer potential insider threats.

During secure design we aim to refine the analysis artifacts into design elements that make the implementation easier and more obvious. We need to provide further insight about the rationale and the mechanics of the trust models and how they integrate into and collaborate with the system. This information is useful to sketch the first version of the system architecture and will assist developers during the implementation phase.

The chapter is organized as follows. Section 3.1 describes a methodology that allows decision-makers to assess the level of trust that can be placed on a cloud provider. Section 3.2 presents a methodology to detect insider threats in a system through the analysis of trust relationships, whereas a methodology and notation to represent trust and reputation requirements is introduced in Section 3.3. Finally, in Section 3.4, we describe a UML profile that allows the specification of trust and reputation models.

3.1 Trust-supported Cloud Sourcing Decision in the Planning Phase

There is an increasing trend to outsource IT services and infrastructures to the cloud (86). This model, also called cloud sourcing[1], is replacing traditional outsourcing engagements due to its advantages (76). These include the provision of elastic IT resources and cost savings as a result of reduced operational costs for complex IT processes (81).

Security and trust are significant barriers for the adoption of clouds in companies (99). Lack of trust in cloud providers lies within the nature of clouds: storage and management of critical data, and execution of sensitive IT processes are performed

[1]Techopedia: `http://www.techopedia.com/definition/26551/cloudsourcing`

beyond the customers control. As a consequence, new security threats arise[2,3], and IT analysts and decision makers must balance the advantages and these threats before making decisions. These decisions range from selecting a cloud provider to determining how much data or which part of the infrastructure to move to the cloud.

It is generally accepted the fact that trust can help in decision-making processes in the absence of complete information (45, 138). Given that information about cloud providers, due to internal policy or strategic reasons, may be uncertain and incomplete, trust can enhance the cloud sourcing decision-making process. This section presents a methodology that evaluates trust in cloud providers and that can help IT analysts to make more informed decisions prior to the sourcing process. The methodology provides a systematic way to gather knowledge about cloud providers and to exploit this knowledge in order to yield trust values that can be used as inputs to the decision-making process. The methodology pinpoints which aspects of the providers should be analysed, indicators that decision makers can use to quantify these aspects, and how these quantifications can be aggregated into trust values. We use trust intervals in order to represent trust and we define a summation operator to aggregate trust intervals. The methodology constitutes a guide that analysts and decision makers can follow to evaluate their trust in cloud providers under several dimensions or viewpoints.

In the following sections, we examine how trust evaluation in the Cloud has been approached in previous works and we describe the trust-aware methodology that we propose. We also apply the methodology to an eHealth case study and summarize some considerations about the methodology as well as lines for future research.

3.1.1 Trust Evaluation in the Cloud

Cloud providers evaluation is a necessary step for cloud sourcing decision-making, but clouds can be evaluated under different angles, including performance (22), scalability (38), accountability (90) and transparency (94).

Traditional evaluation has focused on performance and scalability. For example, Bubak et al. (22) provides an evaluation of Infrastructure as a Service (IaaS) cloud

[2] http://www.infoworld.com/d/security-central/gartner-seven-cloud-computing-security-risks-853

[3] Top Threats to Cloud Computing V1.0,https://cloudsecurityalliance.org/topthreats/csathreats.v1.0.pdf

providers. Their focus is on the performance and ratio cost/performance of different providers for scientific computing in the research. Gao et al. (38) define a set of metrics based on calculating the area of polygons, the vertices of which represent measurable performance and scalability indicators of a cloud service. Assessing accountability and transparency is gaining increasing importance. As an example, Nuñez et al. (90) present a metamodel for the assessment of accountability, whereas Pauly (94) proposes a scorecard for evaluating transparency. In a similar direction, Rak and Aversano (103) discuss a framework for building custom benchmark applications that can be used to evaluate performance of different cloud providers.

The impact of trust for cloud adoption and some trust-related factors that influence users when selecting cloud providers have been identified in previous works (105)(61). In this direction, Sarwar et al. (9) review several works that elicit relevant trust aspects in the cloud. Ahmad et al. (3) argue that trust in the cloud must be built upon a deep knowledge about the cloud computing paradigm and the provider.

In many works, trust depends on the verification of Service Level Agreement (SLA)s (26) or the measurement of QoS attributes (74). Song et al. (121) propose a QoS evaluation model in which key attributes and parameters are defined and quantified. Each attribute is weighted and averaged, yielding a score that can be aggregated with others in the end. These works are usually focused on cloud services evaluation and selection rather than on the cloud providers themselves.

Most existing contributions on trust-aware cloud providers evaluation focus on the technical part of the providers. Only a few of them address social aspects, such as the staff or stakeholders of the provider. For example, Pauley (94) creates a scorecard for evaluating transparency of a cloud provider. Its scorecard includes questions that can be answered by 1 (yes) or 0 (no). Transparency is defined as a combination of other attributes, and the questions address these attributes. In the end, the author sums all the scores and divides them by the total possible. Even when this work addresses some security and privacy issues, it does not evaluate threats and does not consider subjectivity or uncertainty during evaluation.

Pavlidis et al. (96) propose a process for trustworthy selection of cloud providers. This selection is based on how well the cloud provider fulfils the customer's security and privacy requirements. It also aims to reduce uncertainty by justifying trust relationships and by making trust assumptions explicit. Compared to our approach, we consider other

aspects of the cloud providers and we use trust intervals instead of probabilities and weights.

Supriya et al. (72) propose a fuzzy trust model to evaluate cloud service providers that uses the attributes defined by the Service Measurement Index (SMI) (39). Examples of these attributes are assurance, performance and security. Even though uncertainty is embedded in the fuzzy engine, the authors do not provide guidelines on quantifying the attributes or on eliciting cloud knowledge. Qu et al. (102) introduce customers' feedback in the evaluation, although this evaluation is focused on cloud service selection, rather than on cloud provider selection.

Risk as a notion to evaluate cloud services is used by Rödder, Knapper and Martin (111), who propose a set of metrics rather than a methodology. The authors focus on evaluating services offered by the cloud provider, whereas our attention is on the provider as a whole. Metrics weight different attributes and do not include the notion of uncertainty and they do not specify how to retrieve the information required by the metrics, whereas we provide guidelines and a structured knowledge elicitation phase.

Habib, Varadharajan and Mühlhäuser (46) propose an evaluation framework that include hard and soft trust concepts, such as direct interaction, indirect interaction and uncertainty. However, they do not propose a concrete methodology, guidelines for trust factors quantification or a systematic domain knowledge elicitation. In a similar direction, Rehman et al. (126) propose a simple framework for monitoring cloud performance based on user feedback, in which the performance of a cloud service is monitored and predicted by these feedbacks. In this direction, Li et al.(66) propose a taxonomy of performance evaluation concepts. The authors argue that performance evaluation works are difficult to understand and compare because they lack consistency and a common terminology. The taxonomy aims to clarify concepts and inaccurately-used terminology that exist in evaluation works. The same authors suggest that a rigorous methodology for implementing a Cloud evaluation is required, and they propose the Cloud Evaluation Experiment Methodology (CEEM) (64).

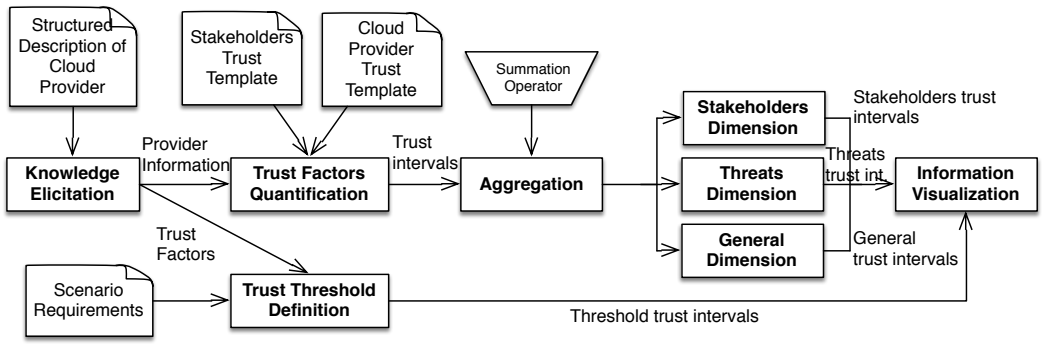
As a conclusion from our literature review, trust has already been incorporated in the evaluation of clouds. However, in most cases, the purpose of this evaluation is service selection, rather than cloud provider selection. Most contributions are also focused on the metrics rather than on a concrete methodology to gather and quantify all the information. Uncertainty or subjectivity, which are intrinsic to the notion of

trust, are usually laid aside. This section aims to fill these gaps. The existing literature provides valuable information about the aspects of cloud providers that are usually considered by cloud customers before moving to the cloud, and our approach, presented in the next section, builds upon this knowledge.

3.1.2 Trust-Aware Methodology

A high-level overview of the methodology is presented in Figure 3.1.

Figure 3.1: Overview of the Methodology



The first step consists of gathering knowledge about the cloud provider. Next, we elicit and quantify a set of trust factors about the provider's stakeholders and about the cloud provider as a whole. In parallel, we specify trust thresholds that are based on the scenario requirements. These thresholds are minimum trust values that we expect for a given scenario. In the following step, the factors are aggregated into three dimensions or viewpoints: a stakeholder dimension, a threat dimension, and a general dimension. In order to perform the aggregation, we define a summation operator. Finally, the information is graphically visualized.

Next we describe each of the steps in more detail.

3.1.2.1 Domain Knowledge Elicitation

The goal of this step is to gather knowledge about the cloud provider and the cloud domain. We suggest context patterns for a structured domain knowledge elicitation (12). These patterns contain a graphical pattern and templates with elements that require consideration for a specific context. In addition, context patterns contain a

method for eliciting domain knowledge using the graphical pattern and templates. For this work we use a specific context pattern, the so-called *cloud system analysis pattern* (11, 14). It describes stakeholders and other systems that interact with the *Cloud*, i.e. they are connected to the cloud by associations. For example, the *cloud provider* offers its resources to *cloud customers* as *Services*, i.e., *IaaS*, *PaaS*, or *SaaS*. However, our methodology is not tied to a concrete methodology for structured domain knowledge elicitation.

3.1.2.2 Trust Factors Quantification

As explained over Chapter 2, trust evaluation models depend on a set of factors that influence trust relationships. These factors must be quantified and aggregated using a trust engine to yield a trust value. This step tackles the identification and quantification of trust factors, whereas Section 3.1.2.4 deals with the trust engine that aggregates these trust factors.

For trust factors identification, we use two different trust templates according to the context patterns methodology (12). The first one is the Stakeholder Trust Template (STT) depicted in Table 3.1, which is a modification over the original stakeholder template (12). This template identifies the trust factors that we consider for the cloud stakeholders, and also in particular, for the staff members of the Cloud provider. The other template is the Cloud Provider Trust Template (CPTT), shown in Table 3.2, which identifies the trust factors that we consider for the cloud provider as a whole. In each table, the first two columns show the name of the factor and its meaning respectively, whereas the last column provides hints for quantifying the factors.

The quantification process in our methodology entails providing two values for each factor: the factor value itself and a confidence value. The latter refers to the confidence that the factor value is accurate. The role of this value is to make explicit the uncertainty derived from having partial and subjective information. For the quantification of both values we decide to use only integer numbers from 0 to 3. More justification on this decision and on the trust engine in general is provided in Section 3.1.4.

In our methodology, threats are sub-factors of two trust factors: *direct interaction* and *3rd party referrals*. The former refers to information about threats derived from previous direct experience with the cloud provider, whereas the latter requires asking external organizations for this information. We use the threats identified by the Cloud

Table 3.1: Stakeholder Trust Template

Direct Interaction	Evaluation of previous direct interaction with the stakeholder.	Analyse the number of incidents and overall satisfaction with the stakeholder in the past.
3rd Parties referrals	Referrals from 3rd parties regarding interactions with the stakeholder.	Ask other organisations about their general satisfaction with the stakeholder.
Knowledge	Stakeholder knowledge on its task.	Check number of years of experience and whether the stakeholder has any certification.
Willingness	Willingness of the stakeholder to perform the task.	Take into account the aforementioned factors; research on the motivations of the stakeholder (e.g. bonuses); check how long it takes him to finish his task.

Security Alliance (CSA), as depicted in Table 3.3, which summarize the experience of a large industrial consortium in the field of cloud computing.

Once we have a factor value and its corresponding confidence value, we calculate a *trust interval* for each factor, as explained in the next definition.

Definition 1 (Trust Interval) *Let v and c be a factor value and its corresponding confidence value, respectively. These values are integer numbers between* 0 *and* 3*. We form the trust interval as:* $TI = [\frac{vc}{3}, \frac{vc}{3} + (3 - c)]$.

This interval is in the domain of the real numbers. 0 and 3 are lower and upper bounds of the interval, respectively. For the rationale of this definition we refer the reader to the contribution by Shakeri et al. (117). Given that we use integer values, there is a finite set of possible intervals during quantification. For example, when the factor value is 2 and the confidence value is 1, the resulting trust interval is $[\frac{2}{3}, \frac{8}{3}]$. Note that when $c = 0$, we have the maximum uncertainty, that is, the interval is $[0, 3]$ and has the maximum width. When $c = 3$, uncertainty is minimum, that is, the interval width is zero because we know the trust value.

Before proceeding to the aggregation of the trust intervals, decision makers define trust thresholds as explained in the next section.

3.1.2.3 Trust Thresholds Definition

This step, which is performed in parallel with the quantification step, defines trust thresholds according to the scenario requirements. These thresholds represent the min-

Table 3.2: Cloud Provider Trust Template

SLA and Contracts	Quality of SLAs and signed contracts that express the conditions and liabilities regarding the service offered by the cloud provider.	Check if there was some abuse of the contract. Do all the interesting services have a well-defined SLA and appropriate for the organization? Are they easy to find and easy to understand?
Security	Provider's concern and actions on security.	Check whether the cloud provider participates in cloud standards bodies such as CloudAudit, Open Cloud Computing Interface, CSA and ENISA. Does the cloud provider perform security assessment? (E.g. COBIT, ISO 27000, NIST SP800-53). Does it offer professional services such as a security assessments of customer environments? Does the cloud provider provide a special email or forum for security/privacy questions?
Long-term viability	Business viability of the cloud provider.	Is the cloud provider profitable? Did the cloud provider have any financial difficulties? How many years in the market? (According to the US Small Business Administration, 50% of businesses fail in the first 5 years). Does the cloud provider have a clear back-up and recovery policy? Are there compensations and are they good in case of problems?
Transparency	Transparency of the provider.	How difficult is to retrieve data from the cloud provider? Does it publish its privacy and security policies? Are they easy to access? Do I know where the data will be located? Does the cloud provider have an open ethical manifest?
Accountability and auditing	How accountable and auditable the provider is.	Does the cloud provider comply with the SAS No.70 Type II, Payment Card Industry Data Security Standard, HIPAA or Sarbanes-Oxley? Is there a clear logging policy? Are there logging systems in place? Is logging information good enough for accountability?
Human Resources Security	Quality and security concern of the employment policies.	Does the provider have a clear, strict, robust hiring policy? (to minimize insider threats; trust stakeholders) Are there procedures for monitoring employees effectiveness? Is there any secure fade-out procedure once a guy leaves the company? Quality of employment: salaries, health insurance. Quality of security trading (how well educated is the staff?).
Direct interaction	Own experience in the interaction with the cloud provider.	Evaluate direct experience against threats.
3rd parties referrals	Referrals from 3rd parties regarding interactions with the cloud provider.	Evaluate 3rd parties referrals against threats.

imum trust that decision makers expect for each trust factor. The goal is to have a yardstick that can be used to check whether cloud providers meet our trust expecta-

Table 3.3: CSA Cloud Threats and their Evaluation

	Description	How to evaluate
Abuse of Cloud Computing (Threat 1)	This threat describes the abuse of the scalable cloud resources, e.g., network connections capacity. For example, the resources can be used by spammers to scale up their operation.	Consider the type, number, and frequency of anomaly checks for cloud computing abuse.
Insecure Interfaces and APIs (Threat 2)	Clouds provide interfaces for provisioning, management, orchestration, and management of services. Security functions, e.g., authentication, and encryption rely upon these. An example for the malicious usage of interfaces is the eavesdropping during clear-text transmission of content.	How often do developers report security issues? How long does it take the cloud provider to fix the problems?
Malicious Insiders (Threat 3)	The cloud provider controls access to the cloud. A cloud customer or end customer has very limited transparency considering data access permissions provided to cloud employees. Hence, the threat of malicious insiders, which are employees of the cloud provider, scales with the resources and offered services in the cloud. An example for a specific problem is policy compliance. Cloud customers or end customers have no visibility into the hiring or monitoring of the cloud providers' employees.	Does the cloud provider publish compliance to laws? Does the provider offer customized SLAs? Does the provider conduct external audits?
Shared Technology (Threat 4)	The different stakeholders in the cloud use the same physical resources, e.g., CPUs and RAM. These are shared using so-called *Hypervisor*s, which provide isolation properties for these physical resources. Side channel attacks on these *Hypervisor*s can provide a stakeholder with inappropriate levels of control of the underlying cloud infrastructure.	Does the provider ensure best practices like penetration testing in preventing successful attacks? Is this documented in a contract?
Data Loss & Leakage (Threat 5)	The threats to data in a cloud scales with the amount of data stored in it. Deletion or alteration of data without a backup is an example. Moreover, cloud databases store data distributed. The links to records in these cloud databases can be destroyed, which results in unrecoverable data.	Are data available about existing incidents of information loss over time? Are information about severity and time to fixing the issue available?
Account or Service Hijacking (Threat 6)	Clouds provide numerous services and credentials, and passwords are often reused. Thus, compromised credentials provide access to a large set of data about activities and transactions of stakeholders. Thus, the attacker can exploit the reputation of a cloud customer and launch a large-scale attack on its end customers. The cloud customer's reputation can lead to directed phishing and farming attacks at its end customers.	Were any identity theft incidents reported or experienced? Is the cloud provider taking responsibility for these incidents or the cloud customers?
Unknown Risk Profile (Threat 7)	Cloud customers and end customers do not own cloud resources. Hence, cloud providers can apply the so-called *security by obscurity* policy. Thus, the cloud customers and end customers do not know the exact specifications of the security mechanisms used in the cloud. This results in an unknown exposure of assets and increases the difficulty of creating a risk profile for a cloud scenario.	List all the data that is not available about the cloud provider? Evaluate the situation based on the number and significance of the missing information.
Unknown Causes (Threat 8)	A significant number of reported cloud problems like outages did not reveal the causes of these problems. In these case we know that incidents happened, but the provider does not release information that allows an analysis. It remains unclear if the problem is addressed properly or even investigated at all.	List all incidents that happened, but the cloud provider did not release enough information to track the incident back to a threat. Hence, this incident cannot be categorized into Threat 1 to 7 and is part of this category. Look at the amount of these incidents and evaluate based on this information.

tions.

For each trust factor, the decision maker assigns an expected factor value and a confidence value. In this case, the confidence value expresses how sure the decision maker is about the need to expect the corresponding factor value. As in the quantification step, for each factor, a trust interval is derived from these values by using Definition 1.

3.1.2.4 Trust Aggregation

During the previous steps we have calculated trust intervals for different factors of stakeholders and cloud providers. This step reduces the number of trust intervals by aggregating them.

Before defining the operator that performs the aggregations, we need another definition.

Definition 2 (Interval Accuracy) *Given a trust interval $[a, b]$, we define the interval accuracy as $IA = 3 - w$, where $w = b - a$ is the width of the interval.*

The maximum possible width of a trust interval is 3 (see Definition 1). When the width is maximum, the interval accuracy is 0 because uncertainty is maximum. On the other hand, when the width of a trust interval is 0, the interval accuracy is 3 because uncertainty is minimum.

Next we define a summation operator that aggregates trust intervals.

Definition 3 (Summation Operator) *Given two trust intervals $[a, b]$ and $[c, d]$, where $a \neq c$ or $b \neq d$, we define the summation operator $\oplus$ as $[a, b] \oplus [c, d] = [e, f]$ where $[e, f]$ is a new trust interval that can be obtained as: $e = \frac{IA_1 a + IA_2 c}{IA_1 + IA_2}$ and $f = \frac{IA_1 b + IA_2 d}{IA_1 + IA_2}$. IA_1 and IA_2 are the interval accuracy of $[a, b]$ and $[c, d]$, respectively. If $a = c$ and $b = d$, then $[a, b] \oplus [c, d] = [a, b] = [c, d]$.*

The resulting interval after a summation is somewhere in between the two source intervals. The uncertainty, represented by the interval accuracy, determines how close e is to a or c, and how close f is to b or d. This is why we weight a, b, c and d by the interval accuracy. The higher the interval accuracy, the more the values of the corresponding interval contributes. Note that the operator has an identity element: $[0, 3]$. This makes sense as this interval expresses the maximum uncertainty and does not add any knowledge to the trust value.

Table 3.4 illustrates some interval summations. Note the last two summations in the table. In the first one, we have complete confidence that the factor value is 0 in the first interval and that the factor value is 3 in the second one, therefore the aggregation yields a value right in the middle. In the last summation, there is complete confidence in the factor value 3, whereas there is limited confidence in the other value. Therefore the aggregation yields a value closer to 3.

Table 3.4: Trust Interval Summations

$[0, 3] + [2, 3]$	$[2, 3]$
$[1, 3] + [2, 3]$	$[1.7, 3]$
$[2, 2.5] + [2.25, 2.5]$	$[2.13, 2.5]$
$[1, 2] + [1, 2]$	$[1, 2]$
$[0, 0] + [3, 3]$	$[1.5, 1.5]$
$[0, 2] + [3, 3]$	$[2.25, 2.75]$

In order to present meaningful trust information, we suggest performing three aggregations that correspond to three dimensions or viewpoints: the stakeholders dimension, the threats dimension and the general dimension. Next subsections explain each of them.

Stakeholders Dimension This dimension illustrates the level of trust in the cloud provider according to the stakeholders working in it. This aggregation is performed by summing all the intervals of all the factors for each stakeholder, and then summing the resulting intervals for all the stakeholders.

Threats Dimension This dimension shows the amount of trust in the cloud provider according to the threats defined by the Cloud Security Alliance (CSA) (see Table 3.3). For each threat, we aggregate the trust intervals of the *direct interaction* and *3rd party referrals* factors.

We believe that having independent trust intervals for each threat is convenient, instead of aggregating all the different threats together, because decision makers can make more fine-grained decisions. For example, if the trust interval is low for the threat *Data Loss & Leakage*, the decision maker can decide not to move the customers data of the organisation to the cloud provider. However, if trust intervals of the other threats for the same cloud provider are high, some services or infrastructures could be outsourced

to that cloud provider. If we aggregated all the threats into a unique trust interval, we would lose this valuable information.

General Dimension This dimension depicts trust in the cloud provider with regards to the rest of trust factors that are not threats, including *Security*, *Transparency* and *Accountability*.

After the trust aggregation step, there are ten trust intervals for a cloud provider: one for the stakeholders dimension, eight for the threats dimension (i.e. one for each threat) and one in the general dimension.

3.1.2.5 Trust Information Visualization

The last step consists of plotting the trust intervals for each dimension for comparison purposes and decision making.

In the Y-axis, we represent possible trust values, whereas in the X-axis we represent the three dimensions. For each dimension, we draw a line from the lower bound to the upper bound of its trust intervals. This arrangement allows fast comparison between providers in each dimension. Likewise, it allows comparing the trust intervals with the trust thresholds.

This is better illustrated in the next section, where we apply the methodology to an eHealth scenario.

3.1.3 Application Example: eHealth

In this section we present an application of our methodology to a case study provided by the EU project Network of Excellence on Engineering Secure Future Internet Software Services and Systems (NESSoS)[4]. The scenario concerns managing Electronic Health Record (EHR)s in clouds. EHRs contain any information created by health care professionals in the context of the care of a patient. Examples are laboratory reports, X-ray images, and data from monitoring equipment.

Security concerns in this scenario include:

- Confidentiality of EHRs in communication and storage
- Data separation of EHRs and other data of the eHealth applications

[4]The NESSoS project: `http://www.nessos-project.eu`

- Authentication mechanism to guarantee confidentiality and integrity
- Availability of EHRs
- Availability of network connection
- Data Origin Authentication

Given these security concerns, the CSA threats that become more relevant are the following: *Insecure Interfaces and APIs (Threat 2)*, because these are essential for security functionalities like authentication; *Malicious Insiders (Threat 3)*, because they could steal EHRs and use them for blackmailing or similar criminal activities. *Shared Technology (Threat 4)* and, specially, *Data Loss & Leakage (Threat 5)*, can lead to a loss of confidentiality of EHRs or data separation. *Account or Service Hijacking (Threat 6)* leads to bypass authentication controls, including those for data origin authentication; *Unknown Risk Profile (Threat 7)* and *Unknown Causes (Threat 8)*[5] can also have a negative effect on all the security concerns.

For the evaluation, we consider the following cloud vendors: Amazon, Apple, Microsoft and Google. We lay stakeholders evaluation aside and we focus on evaluating trust in the threat and general dimensions; retrieving stakeholders information is more difficult but the process and the aggregation would be similar. Next subsections include each step in our methodology.

Trust Factor Quantification and Thresholds Definition Threats quantification is based on a data set from CSA, which mapped 11 491 cloud security incidents to these threats[6].

As explained before, for each trust factor, including the threats, we assign a factor value and a confidence value. For example, in the case of *Threat 1* for Amazon, we assigned factor value 0 and confidence value 2. The rationale, which must also be included as part of the analysis, is that we found three incidents on record and one that had a significant amount of user accounts affected. As another example, for *Security* trust

[5]Note that the original CSA Top Threats are just 7, but the CSA documented cloud security incident referenced numerous incidents that cannot be categorized because of a lack of information. This lead us to adding an additional threat.

[6]Documented Cloud Security Incidents: `https://cloudsecurityalliance.org/download/cloud-computing-vulnerability-incidents-a-statistical-overview/`

factor in Microsoft, we assigned factor value 3 and confidence value 2. The rationale is that Microsoft considers some certifications (e.g. ISO 27001) and complies with the CSA control matrix and FedRAMP. Applying Definition 1, we obtain the trust interval $[0, 1]$ for the first example, and $[2, 3]$ for the second example. Table 3.5 shows the whole quantification for one of the cloud vendors.

In parallel and based on the security requirements of the scenario, we define minimum trust values for each trust factor. These thresholds, already aggregated in the threat and general dimensions, are presented in Table 3.6.

Trust Aggregation We aggregate the trust intervals of every factor for a given cloud provider. As an example, consider the following: Apple has trust interval $[0, 2]$ for *Security* and $[0.33, 2.33]$ for transparency. We use the operator in Definition 3 to aggregate these intervals. The resulting interval is $[0.17, 2.17]$. We would now aggregate this trust interval with the one corresponding to *Accountability and Auditing*, and so forth, until we reach a final trust interval in the general dimension. The resulting trust interval in the general dimension for each cloud provider is shown in the last column of Table 3.7.

We assume that we have no direct previous experience with the providers. Therefore, there is no need to aggregate trust intervals in the threat dimension, which this time only considers information from *3rd party referrals*, in this case, from CSA. Trust intervals for each threat and cloud provider are presented in Table 3.7.

Trust Visualization Figure 3.2 shows the trust intervals of all cloud providers, whereas Figure 3.3 compares the trust intervals with the trust thresholds.

As a conclusion, we see in Figure 3.3 that no cloud provider upholds all trust thresholds. If we focus on data loss and leakage (Threat 5), we see that none of them fully upholds it, but Apple and Amazon seem more interesting under this lens. Microsoft seems to be the best cloud provider according to general properties such as security or transparency, followed by Google. If we focus on the threats in general, we note that again, no cloud provider performs well for all threats. However, according to our analysis, the less bad performers are Microsoft, Apple, Amazon and Google in this order.

Summing up, if we were analysts, we would either not pursue any cloud provider for our scenario at this time and repeat the analysis later, or would confront the cloud

Table 3.5: Trust Factors Quantification for Microsoft

Cloud Provider	Trust Factor	Sub- trust factor	Factor value	Confidence value	Rationale
Microsoft	3rd Party referral	Threat 1	3	2	No incidents on record and there an assumption that the provider invests a lot in controls to prevent those.
Microsoft	3rd Party referral	Threat 2	1	1	Three incidents on record and one has affected a medium amount user accounts affected (14000).
Microsoft	3rd Party referral	Threat 3	2	2	One incident on record and other cloud provider do not seem to report incidents at all. Assumption the provider is transparent.
Microsoft	3rd Party referral	Threat 4	2	1	Two incidents on record and it is assumed they caused limited affects.
Microsoft	3rd Party referral	Threat 5	1	2	Three incidents on record and it is assumed the cloud provider has problems improving its security controls.
Microsoft	3rd Party referral	Threat 6	1	2	No incidents on record. No information about security controls for this threat available.
Microsoft	3rd Party referral	Threat 7	2	2	Once incident on record with only limited effect.
Microsoft	3rd Party referral	Threat 8	0	2	Eleven incidents reported and the trend seems to increase. This is the largest problem of the provider.
Microsoft	Security	-	3	2	Certifications exist for ISO 27001,SOC,CSA control matrix, FedRAMP. Rumors state that more should follow.
Microsoft	Transparency	-	2	2	We saw texts about identity management and access control on the homepage.
Microsoft	Accountability	-	0	1	We did not find any information about this on the homepage.
Microsoft	SLA	-	2	2	Availability is guaranteed with 99.99%. We could not find rumors about major outages.
Microsoft	Human Resources Security	-	0	1	No information was made public on the homepage and we did not hear rumours.

Table 3.6: Trust Thresholds

Threat 1	Threat 2	Threat 3	Threat 4	Threat 5	Threat 6	Threat 7	Threat 8	General
[1.0,1.0]	[0.67,2.67]	[0.33,2.33]	[2.0,2.0]	[2.0,2.0]	[0.67,2.67]	[0.33,2.33]	[0.33,2.33]	[0.67,2.67]

Table 3.7: Trust Intervals for Cloud Providers

	Threat 1	Threat 2	Threat 3	Threat 4	Threat 5	Threat 6	Threat 7	Threat 8	General
Amazon	[0,1]	[0.67,1.67]	[0,3]	[0.33,2.33]	[1.33,2.33]	[0,3]	[0.33,2.33]	[0,1]	[0.34,1.86]
Apple	[0.67,1.67]	[0,1]	[0,0]	[0.33,2.33]	[1.33,2.33]	[0.33,2.33]	[0.67,1.67]	[0.67,2.67]	[0.02, 2.02]
Microsoft	[2,3]	[0.33,2.33]	[1.33,2.33]	[0.67,2.67]	[0.67,1.67]	[0.67,1.67]	[1.33,2.33]	[0,1]	[0.8, 2.25]
Google	[1.33,2.33]	[0,1]	[0.33,2.33]	[1.33,2.33]	[0,1]	[0.33,2.33]	[0.33,2.33]	[0,1]	[0.53, 2.39]

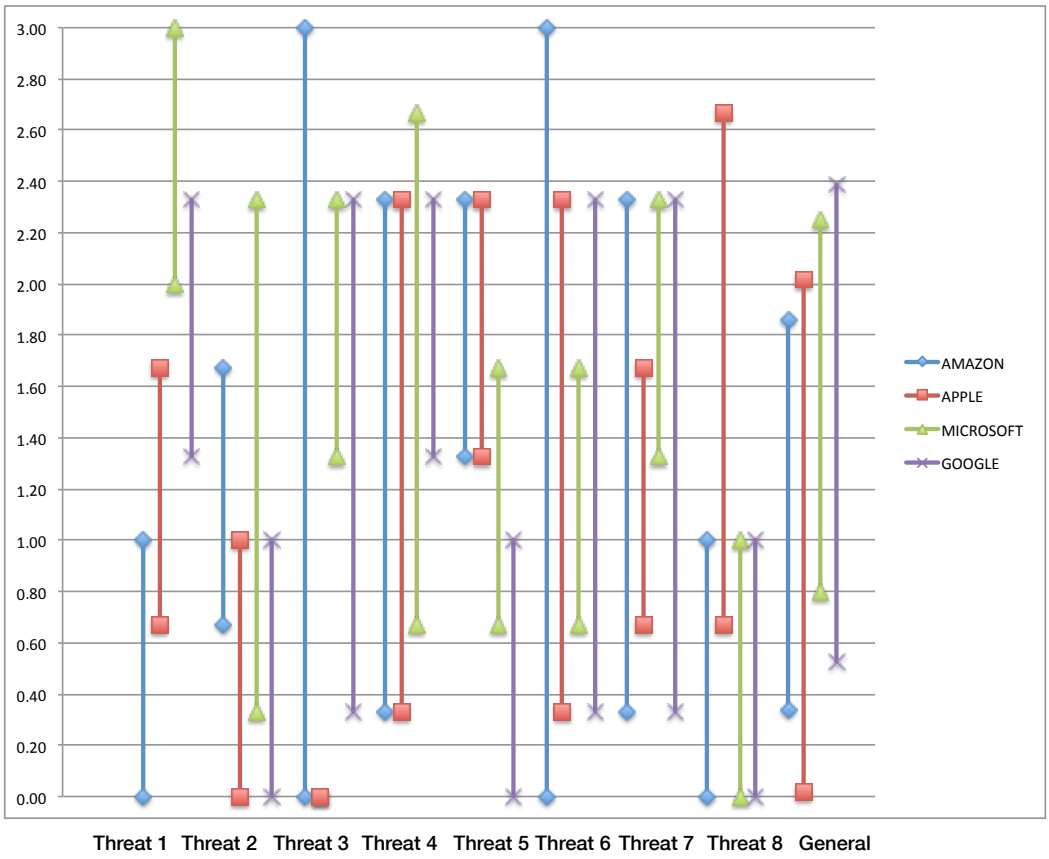
providers with the results and ask for a detailed justifications for their security mechanisms, especially regarding threat 5.

3.1.4 Discussion

As discussed in Chapter 2, there are many trust and reputation engines in the literature (56). Given that this methodology is aimed at analysts and decision makers, who do not necessarily have much mathematical background, a requirement for our trust engine was its simplicity. The engine that we present in this work uses trust intervals to represent trust information. There are other engines that are easier to use, such as summation or average engines. However, they present two main problems. First, they usually require weighting the attributes, and selecting weights is difficult and prone to trial-and-error mechanics. Second, they lack the capability to represent uncertainty, which is a concept highly coupled to the notion of trust. We believe that trust intervals present a good trade-off between simplicity and expressiveness.

Best practices in risk assessment indicate that practitioners should set an even number of choices since users tend to choose the middle value in odd numbered scales (92). This is why we quantify each trust factor with 4 possible values (i.e. from 0 to 3). We think that 2 would give too few flexibility, whereas more than 4 would be confusing.

Figure 3.2: Comparison of Trust Intervals for the selected Cloud Providers

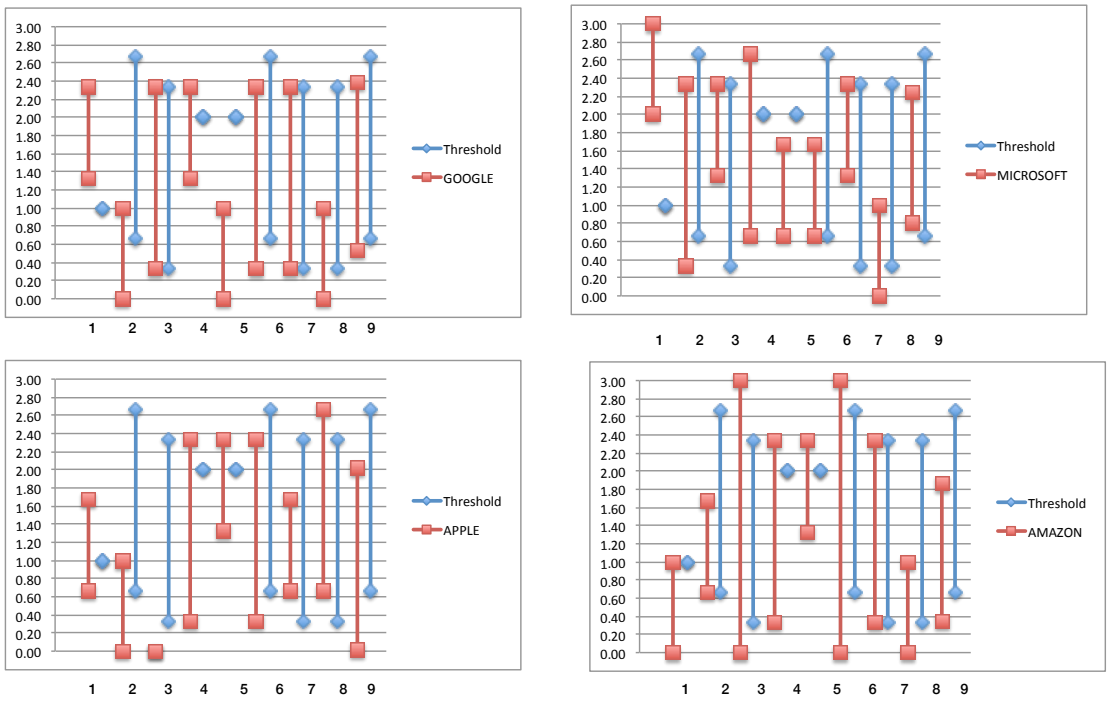


A disadvantage of our methodology is that it relies on data that in many cases may not be accessible or available. Cloud providers may be reluctant to provide certain information and it might not be straightforward to gather knowledge about the stakeholders of a cloud provider.

Another source of imprecision is subjectivity. By definition, trust is subjective and therefore some of the information that the methodology requires may have a subjectivity bias. The results of the trust evaluation may not be completely accurate, but we advocate that even minimal or partially subjective information is better than blind decision-making. In order to avoid strong subjectivity bias, it is important to state the rationale for each factor quantification.

Subjectivity draws a line between trust and trustworthiness. Having a trustworthiness value would help in determining trust. Whereas trust usually depends on subjective

Figure 3.3: Contrasting Trust Thresholds and Trust Intervals



information and may change among trustors, trustworthiness is an objective measure of many different qualities. The ideal situation occurs when trust in a trustee matches the trustworthiness of that trustee (95). This is the reason why we claim that we are evaluating trust and not trustworthiness.

3.2 Trust-supported Threats Analysis

As reported by the 2014 CyberSecurity Watch Survey, 28% of cybercrimes were committed by insiders (120), and 46% of the respondents thought that damage caused by insider attacks was more severe than damage from outsider attacks. In fact, insider attacks can cause significant damage to the affected organizations e.g loss of money, loss of reputation, or loss of customers, among others.

The CERT Insider Threat Center of the Software Engineering Institute (118) defines an insider as "a current or former employee, contractor, or business partner who has or had authorized access to an organization's network, system, or data and in-

tentionally exceeded or misused that access in a manner that negatively affected the confidentiality, integrity, or availability of the organization's information or information systems". Insider attacks are more difficult to detect because they come from trusted employees who have legitimate and often privileged access to critical or valuable assets, and have knowledge of the organization and its processes. The effective defense from insider attacks call for preventive measures that detect and assess the risks associate with insiders, rather than for reactive measures after the attack has been conducted.

In this section, we present an approach to assist security engineers in the detection of insider threats during the analysis phase of the system development life cycle. Our approach is complementary to other threats identification approaches that rely on the analyst level of expertise such as risk assessment (70). With our approach, the security engineer can identify automatically the insider threats that exist in a given organization and permission setting and assess the associated risks.

The approach consists of first modelling the actors[7] (i.e. the system stakeholders), their goals, their assets, the security properties (e.g confidentiality, integrity, availability) that stakeholders want to hold for their assets, the permissions that the stakeholders have on assets, and delegation and trust of permissions relationships among them. Trust of permission relationships represent the belief of the grantor of a permission on an asset that the grantee will not misuse it: an actor can be either trusted with a permission or distrusted. The level of *trust* associated with an agent with respect to a granted permission is crucial to assess the risk of the agent being an insider threat: the lower the level of trust associated with a permission is, the higher is the likelihood that the agent will misuse the permission according to the trustor's perception. For this modeling, we use the SI* requirements modeling language (77), because it targets socio-technical systems.

In order to support the automatic detection of insider threats, we extend the SI* requirements modelling language proposed in (7) with an *asset* and *trust* model. The asset model associates assets with a *sensitivity* value that represent how valuable the asset is for the owner. The trust model extends the native binary SI* trust model (trusted, not trusted) and allows associating different levels of trust (e.g. high, medium

[7]We use the term *actor* instead of *entity* because we are using the nomenclature of SI*. However, according to our conceptual model presented in Section 2.2, an actor is an entity, which can be a trustor or a trustee.

and low trust) with a permission granted to an agent. Based on the sensitivity and trust levels, we define a set of rules to automatically identify insider threats to an asset and prioritize them based on the risk associated with the threat. The risk associated with the insider threat is given by both the likelihood that the threat occurs and the cost of the permission being misused. The former is quantified by the *trust level* associated with the permission granted to the insider agent, whereas the latter is quantified by the *sensitivity* of the asset being harmed.

The rest of the section is organized as follows. First, we introduce the SI* framework and its extensions proposed in (7). Next, we present the asset and the trust model followed by the process to identify and prioritize insider threats. Finally, we apply the approach to an eHealth case study of patient monitoring.

3.2.1 The SI* Framework

The SI* modeling language (77) has been proposed to capture security and functional requirements of socio-technical systems. SI* is founded on the concepts of *agent*, *role*, *service*, and relations such as *And/Or decomposition* and *means-end*.

An *agent* is an active entity with concrete manifestations and is used to model humans as well as software agents and organizations. A *role* is the abstract characterization of the behaviour of an active entity within some context. An *actor* is the general way of referring to agents and roles when we do not need to distinguish. The term *service* is used to denote a *goal*, a *task* and a *resource*. A *goal* captures a strategic interest that is intended to be fulfilled. A *task* represents a particular course of actions that produces a desired effect. It can be executed to satisfy a goal. A *resource* is an artifact produced/consumed by a goal or a task. *And/Or decomposition* is used to refine a goal, while *means-end* identifies goals that provide means for achieving another goal or resources produced or consumed by a goal or task.

SI* also captures social relationships (e.g., *delegation* and *trust*) for defining the entitlements, capabilities and objectives of actors. Originally, a *delegation* marks a formal passage of responsibility (*delegation execution*) or authority (*delegation permission*) from an actor (*delegator*) to the actor receiving the responsibility/authority (*delegatee*) to achieve a goal or to provide a resource.

Trust in SI* comes in two flavours. Trust of execution represents a relation between two actors representing the expectation of one actor (*trustor*) about the capabilities of

the other (*trustee*). Trust of permission is represented as the expectation of the trustor about the behaviour of the trusteee with respect to the given permission.

Asnar et al. (7) extends SI* to represent different types of actors' permissions on resources (Table 3.8) and different types of relationships between resources (Table 3.9). Goals and resources are considered as assets that need to be protected because they bring value to organizations. In order to specify how an asset needs to be protected, we use the concept of *security requirement* defining a specific security property, such as confidentiality, integrity, and availability. The permission type granted on a resource determines the type of actions an actor can perform on a resource, as depicted in Table 3.8. Actors than own a resource has the *manage* permission type on that resource. Actors that have the *manage* permission type on a resource automatically inherit the *modify* and *access* permission types on that resource. Likewise, actors that have the *modify* permission type on a resource inherit the *access* permission type on that resource.

A permission type might yield to the violation of a specific security property if the actor misuses the actions granted by the permission type. Moreover, a given permission granted on a resource can be extended to other resources that are related to the resource by the relations reported in Table 3.9.

Table 3.8: Permissions on Resource

Permission Type	**Description**	**(Possible) Affected Sec. Property**
Access (low-level)	Actor only has the permission to access/read/use the resource.	Confidentiality
Modify (medium-level)	Actor can change the content of the resource.	Integrity
Manage (high-level)	Actor has the permission to modify the resource, delegate permissions to other actors and modify permissions to other actors.	Availability

In order to allow the formal analysis of SI* models, the semantics of SI* is defined in the Answer Set Programming (ASP) paradigm, which is a variant of Datalog with negation as failure and disjunction. This paradigm supports specifications expressed in terms of facts and Horn clauses, which are evaluated using the stable model semantics.

Table 3.9: Relationships between Resources

Relationship	Description
store_ in	An informational resource is stored in a physical resource.
part_ of	A resource consists of other resources.
require	A resource might require another resource to function.

Here, SI* models are encoded as sets of facts. Rules (or axioms) are Horn clauses that define the semantics of SI* concepts. To support the formalization in ASP, the DLV inference engine (63) is used. Table 3.10 summarizes the predicates to formalize an SI* model in ASP.

We perform two extensions over SI*: an asset model and a trust model, which are described in the next sections.

3.2.2 Asset Model

An asset is a service for which the owner specifies the sensitivity and a security property that expresses the need of protecting the service. We introduce the predicates shown in Table 3.11.

In our model, we distinguish between two types of assets: *direct* and *indirect* assets. *Direct assets* are services for which a security property and a sensitivity level are explicitly modeled in the SI* model, while *indirect assets* are services for which the security property and the sensitivity level is determined based on the relations with other services. The identification of indirect assets is based on a set of rules, reported in Table 3.12, that considers the relations among resources (i.e. *stored_ in*, *part_ of* and *require*) and the relationship *means_ end* among the resources and the goals that requires the resources to be fulfilled. We assume that if an asset is related to a service by one of these relations, the same security property should hold for the asset and service (axioms $S1$-$S5$) and that the two assets should have the same sensitivity level (axioms $S6$-$S9$). If the direct asset is related to another direct asset only the security property is propagated to the other asset.

Table 3.10: Predicates for ASP SI* Formalization

Goal model
service(Service:s)
goal(Goal:g)
resource(Resource:r)
actor(Actor:x)
agent(Agent:a)
role(Role:p)
play(Agent:a,Role:p)
provide(Actor:a, Goal:g)
own(Actor:a, Goal:g)
own(Actor:a, Resource:r)
subgoal(Goal g1, Goal:g)
means_ end(Resource:r, Goal:g)
means_ end(Goal:g, Resource:r)
Resource model
stored_ in(Resource:r, Resource:r1)
part_ of(Resource:r, Resource:r1)
require(Resource:r, Resource:r1)
Permission model
permission(Actor:a, Resource:r, PType:pt)
del_ perm(Actor:a, Actor:a1, Resource:r, PType:pt)
trust_ perm(Actor:a, Actor:a1, Resource:r)
Security requirements and Threats model
secure_ req(Resource:r, SProperty:sp)
secure_ req(Goal:g, SProperty:sp, Resource:r)
threat(Actor:a, Resource:r, SProperty:sp)
threat(Actor:a, Goal:g, SProperty:sp, Resource:r)

3.2.3 Trust Model

The SI* trust model only supports binary trust values: either an agent is trusted or is distrusted for a given permission on a resource. However, in real scenarios trust is not a binary value but an agent can be assigned different levels of trust. As explained in Chapter 2, a trust relationship holds between two entities: a trustor (the one who

Table 3.11: Predicates for the Asset Model

Predicate	Meaning
sec_ req(s,sp,p)	Security property *sp* should be preserved for a service *s* owned by a role *p*.
service_ instance(s, a, p)	Instance of service *s* owned by agent *a* who plays role *p*.
sec_ req(service_ instance(s, a, p),sp,a,p)	Security property *sp* should be preserved for a specific instance of a service *s*.
sensitivity(s,sl,p)	Service *s* owned by role *p* has sensitivity level *sl*.
sensitivity_ instance(service_ instance(s, a, p),sl,a,p)	Associates a sensitivity level *sl* to an instance of the service *s* owned by the agent *a* playing the role *p*.
asset(s, p)	Service is an asset, where *s* is a service and *p* is the role who owns it.
asset_ instance(service_ instance(s, a, p),a,p)	Instance of service *s* is an asset owned by agent *a* playing the role *p*.

places trust) and a trustee (the one performing a given action and to who(m) the trustor places trust in). The context in which this trust relationships takes place in this case is a permission that is granted to the trustee on a given asset. The trust level is important as it indirectly provides information about the likelihood (as perceived by the trustor) that the trustee will misuse the granted permission to harm the asset.

The trust model that we propose associates a *trust level* with a trust of permission relation between two agents. The trust levels are then translated into trust labels that are used to define insider threats identification rules, which determine if an agent may misuse a granted permission on an asset and the risk associated to the threat.

We assume that trust levels can be represented in two forms: numbers in the interval $[0, 1]$ and qualitative labels such as *Very Good*, *Good*, *Neutral*, *Bad*, *Very Bad*, as proposed by Agudo, Fernandez-Gago and Lopez (2). We assume that some trust values are already assigned to trust of permission relationships between agents in the SI* model and that these values are leveraged by the organization stakeholders in order to compute trust values for pairs of agents for which such relationships are not explicitly modelled. To determine the trust level that an agent A places on another agent B regarding how B will behave with respect to a granted permission, we leverage the trust of permission relationships that other agents have with A and B. Therefore, according

Table 3.12: Axioms for Identifying Indirect Assets

S1	$sec\_req(R, SP, P) \leftarrow$ $store\_in(R1, R) \wedge sec\_req(R1, SP, P)$
S2	$sec\_req(R1, SP, P) \leftarrow$ $part\_of(R1, R) \wedge sec\_req(R, SP, P)$
S3	$sec\_req(R, SP, P) \leftarrow$ $require(R1, R) \wedge sec\_req(R1, SP, P)$
S4	$sec\_req(G, SP, R, P) \leftarrow$ $secure\_req(R, SP, P) \wedge means\_end(G, R)$
S5	$sec\_req(G1, SP, R, P) \leftarrow$ $subgoal(G1, G) \wedge sec\_req(G, SP, R, P)$
S6	$sensitivity(R, SL, P) \leftarrow$ $store\_in(R1, R) \wedge sensitivity(R1, SL, P)$
S7	$sensitivity(R, SL, P) \leftarrow$ $part\_of(R1, R) \wedge sensitivity(R1, SL, P)$
S8	$sensitivity(R, SL, P) \leftarrow$ $require(R1, R) \wedge sensitivity(R1, SL, P)$
S9	$sensitivity(G, SL, P) \leftarrow$ $means\_end(G, R1) \wedge sensitivity(R1, SL, P)$

to the classification performed in Section 2.2.2, we propose the integration of a trust propagation model into SI*. As explained in Section 2.2.3, propagation models require two operators: a concatenator operator and an aggregator operator. Before defining these operators though, we must introduce the concept of trust statement.

Definition 4 (Trust Statement) *A trust statement is an element* $(Trustor, Trustee, Context, Value) \in E \times E \times C \times TD$*, where* E *is the set of all entities in the system;* C *is a set representing a context; and* TD *is a Trust Domain.*

Trust of permission relationships are a particular instance of trust statements where $Context$ is the permission granted to the trustee on an asset instance and $Value$ is the trust level placed by the trustor in the trustee for the permission. To represent trust statements we introduce the predicate *trust_ perm_ instance(A, A1,asset_ instance (service_ instance(S,A,P)), PT, TL)*, which essentially means that agent A trusts agent $A1$ with a level TL to not misuse service S with permission type TL and owned by A.

Trust statements can form trust chains, and the concatenator and aggregator operators evaluate trust over these chains, as defined next.

Definition 5 (Concatenator Operator) *A concatenator operator is a function,* $f : \bigcup_{n=2}^{\infty} \overbrace{TD \times \cdots \times TD}^{n} \longrightarrow TD$*, that calculates the trust level associated to a path or chain of trust statements, such that* $f(v_1, \ldots, v_n) = 0$ *if, and only if,* $v_i = 0$ *for any* $i \in \{1, \ldots, n\}$*, where* $v_i \in TD$ *and* TD *is a trust domain.*

Definition 6 (Aggregator Operator) *An agregator operator is function is used to calculate the trust level associated to a set of paths or chains of trust statements. It is defined as,* $g : \bigcup_{n=2}^{\infty} \overbrace{TD \times \cdots \times TD}^{n} \longrightarrow TD$*, where* TD *is a trust domain and*

1. $g(z_1, \ldots, z_{i-1}, z_i, z_{i+1}, \ldots, z_n) = g(z_1, \ldots, z_{i-1}, z_{i+1}, \ldots, z_n)$ *if* $z_i = 0$
2. $g(z) = z$

As a consequence of applying these operators to SI* models, an agent might end up holding several trust of permission relations with a given agent. However, it would be optimal if an agent only holds one value for any other agent of the system. Resolution functions could solve this.

Definition 7 (Trust Resolution Function) *A trust resolution function is a function,* $f : \bigcup_{n=2}^{\infty} \overbrace{TD \times \cdots \times TD}^{n} \longrightarrow TD$*, such that* $f(v_1, \ldots, v_n) \leq max(v_1, \ldots, v_n)$ *and* $f(v_1, \ldots, v_n) \geq min(v_1, \ldots, v_n)$*, where* $v_i \in TD$ *and* TD *is a trust domain.*

Basically, given a set of trust values, the resolution function produces one unique representative trust value that is upper bounded by the maximum and lower bounded by the minimum of the original trust values. Operators and the resolution function rely on functions to compute the trust values. For this purpose, different functions could be used, like the maximum, minimum, arithmetic or geometric means, etc.

Once we obtain the final numeric values for every trust of permission relationship, transformation rules must be used in order to translate these values, which are in an interval $[a, b]$ ($[0, 1]$ is the chosen one in this case) into a label in a given set of labels that forms a trust scale (2).

The next section explains the threat model that we consider in this approach

3.2.4 Threat Model

We assume that an agent A is an insider for a given asset S when two conditions hold:

a) A is granted a permission PT on the asset S that is sufficient to violate the security property associated with S;

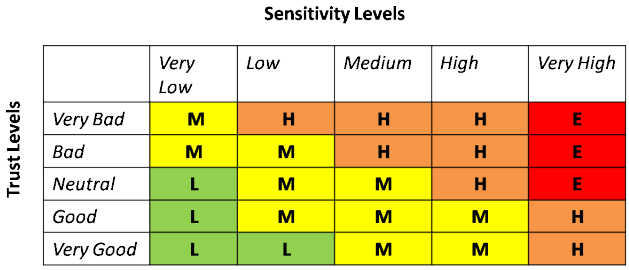
b) The agent who owns the resource S does not fully trust A with permission PT.

We consider that the severity of a threat depends on the sensitivity levels of assets and the trust levels on the trust of permissions relationships. We introduce a *threat* predicate to specify when an agent is an insider for a given instance of an asset and the risk associated with the insider threat. Figure 3.4 is an example of how the risk level of a threat can be determined based on sensitivity and trust levels.

The identification of the insider threats and their risk level is based on a set of axioms reported in Table 3.13, where we list the axioms to detect insider threats to assets' confidentiality, integrity and availability with extreme severity level.

The modeling and the reasoning based on the above axioms are supported by the SI* tool which is an Eclipse plug-in equipped with a DLV engine. The tool interface allows to draw an SI* model which is automatically translated into ASP specification. The

Figure 3.4: Threats Severity Levels



The rows of the table represent the trust levels, while the columns represent the sensitivity levels. Each entry of the matrix specifies the severity level for a given combination of sensitivity and trust levels. The severity level can assume one of the following values: Low, Moderate, High, Extreme. How trust and sensitivity relates to each other depends on the organization's policy and should not be fixed beforehand. Intuitively, the higher the sensitivity of an asset, the higher the damage for the organization. Similarly, the higher the trust level, the lower the likelihood that the agent will misuse the granted permission

tool also allows to input the rules for insider threat identification so that the problem of identifying insider threats is the same as checking a DLV program that formalize the SI* model and the axioms.

The next section apply the process to an eHealth monitoring scenario.

3.2.5 Application Example: eHealth

To illustrate our approach, we use a patient monitoring scenario from the eHealth case study proposed in the NESSoS European project[8]. The scenario involves five main actors. **Patient** is monitored by a smart T-shirt which measures medical data (e.g., heartbeat rate, blood pressure, etc.) and transfers them to the Hospital's computer system. When the patient's condition is abnormal, the doctor makes a diagnosis and produces a prescription. The patient receives his prescription and requests the drug delivery service to the pharmacy. The **Hospital** provides medical services to patients. The hospital monitors patients' health and manages patients' data, which are stored in the hospital's computer. When the patient has some problems, the hospital assigns a doctor to diagnose the patient. The **Pharmacy** is responsible for managing drugs and provide them to the patients. All the information about drugs is stored in the pharmacy's computer. The **Pharmacist** works for the pharmacy and is responsible for

[8] http://www.nessos-project.eu/

Table 3.13: Axioms for Identifying Insider Threats

Insider Threat to Confidentiality
a *threat(A1,asset_ instance(service_ instance(S, A, P), A, P) , confidentiality,* **extreme***)* ← *asset_ instance(service_ instance(S, A, P), A, P)* ∧ *sec_ req_ instance(service_ instance(S, A, P), confidentiality, A, P)* ∧ *permission_ instance(A1, service_ instance(S, A, P), access)* ∧ *sensitivity_ instance(service_ instance(S, A, P),* **very high***, A, P)* ∧ *trust_ perm_ instance(A, A1, asset_ instance(service_ instance(S, A, P), access,* **very bad***)* ∧ *A1* ≠ *A*
b *threat(A1,asset_ instance(service_ instance(S, A, P), A, P) , confidentiality,* **extreme***)* ← *asset_ instance(service_ instance(S, A, P), A, P)* ∧ *sec_ req_ instance(service_ instance(S, A, P), confidentiality, A, P)* ∧ *permission_ instance(A1, service_ instance(S, A, P), access)* ∧ *sensitivity_ instance(service_ instance(S, A, P),* **very high***, A, P)* ∧ *trust_ perm_ instance(A, A1, asset_ instance(service_ instance(S, A, P), access,* **bad***)* ∧ *A1* ≠ *A*
c *threat(A1,asset_ instance(service_ instance(S, A, P), A, P) , confidentiality,* **extreme***)* ← *asset_ instance(service_ instance(S, A, P), A, P)* ∧ *sec_ req_ instance(service_ instance(S, A, P), confidentiality, A, P)* ∧ *permission_ instance(A1, service_ instance(S, A, P), access)* ∧ *sensitivity_ instance(service_ instance(S, A, P),* **very high***, A, P)* ∧ *trust_ perm_ instance(A, A1, asset_ instance(service_ instance(S, A, P), access,* **neutral***)* ∧ *A1* ≠ *A*

Axioms $T1.a$ - $T1.c$ identify insider threats to confidentiality: the insider has *access* permission on the asset being harmed and the owner of the asset places *very bad*, *bad* or *neutral* trust level in the insider for the granted permission.

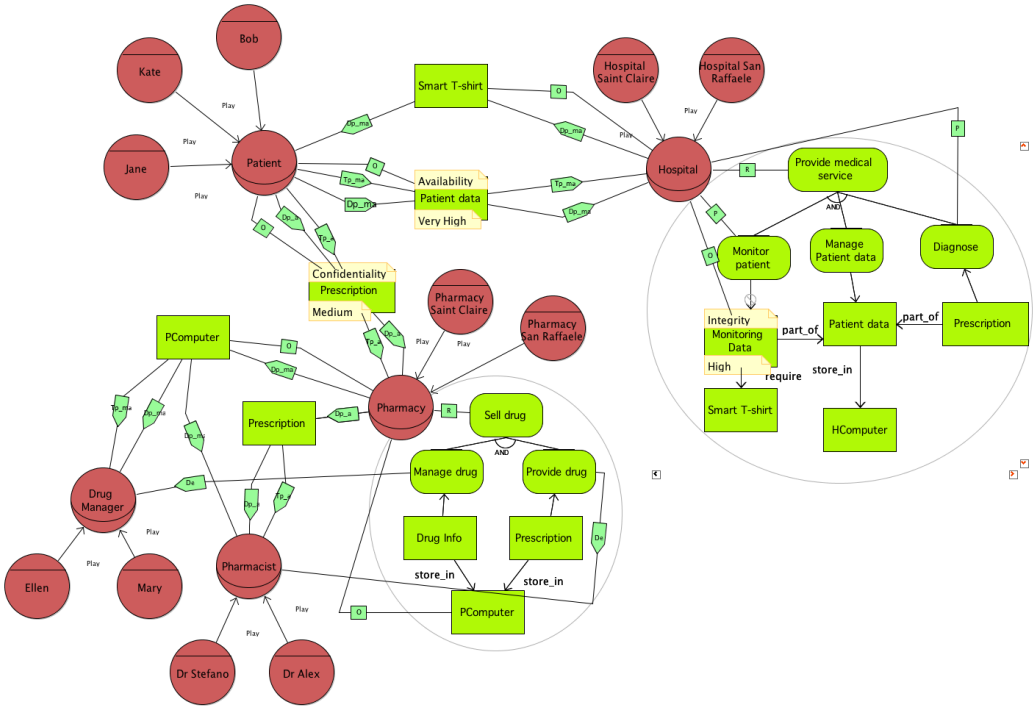
providing drugs to be delivered according to the prescription received from the patient. The prescription information is stored in the pharmacy's computer. Finally, the **Drug manager** works for the pharmacy and is responsible for managing the drugs. All the drugs' information is also stored in the pharmacy's computer.

Figure 3.5 shows the SI* model for this scenario. The model consists of five roles: the Hospital, the Patient, the Pharmacy, the Pharmacist, and the Drug Manager. In this particular example, we assume that Patient (*Role*) can be played by three agents Bob, Kate, and Jane. The Patient (**O***wns*) the resources Patient data and Prescription. It delegates to the Hospital the manage permission on Patient data, and it delegates the access permission on Prescription to the Pharmacy. The Pharmacy has the intention (**R***equest*) to fulfill the goal Sell drug which is (*AND-decomposed*) into subgoals Manage drug and Provide drug: the fulfillment of Manage drug is delegated to the Drug Manager while the fulfillment of Provide drug is delegated to the Pharmacist. The Pharmacy (**O***wns*) the resource PComputer. It grants to Drug Manager the manage permission on PComputer and the access permission on Prescription to the Pharmacist. The Hospital (*Role*) has an intention (**R***equest*) to fulfill the goal Provide medical service which is (*AND-decomposed*) into subgoals Monitor patient, Manage patient data, and Diagnose. Some goals can produce or consume resources. For example, the goal Diagnose requires the resource Patient data and produces the resource Prescription. The Hospital (**O***wns*) the resource Smart T-shirt and delegates to the Patient the manage permission on it. Table 3.14 depicts a snapshot of the formalization of the model in ASP.

Now we proceed to the identification of critical assets. There are three direct assets owned by the Patient role: Prescription, Patient Data, and Monitoring Data. The Patient requires *confidentiality* to hold for Prescription, *availability* should hold for Patient Data, while *integrity* should be satisfied for Monitoring Data. Smart T-Shirt, HComputer, PComputer, Diagnose, Manage Patient data, Monitor Patient, and Provide Drug are indirect assets. For example, PComputer is an indirect asset because the asset Prescription is stored in PComputer and thus the *confidentiality* of PComputer needs to be preserved. Similarly, the goal Diagnose is an indirect asset because it is linked to the asset Prescription by a means_end relation, and thus also the confidentiality of the goal needs to hold.

Next we determine permissions on assets. This step determines the permissions that roles are granted on assets. The permissions are assigned to roles based on a

Figure 3.5: Patient Monitoring Scenario in SI*



The circles denote roles or agents, the ovals denote goals, while the rectangles represent resources. **Dp_a**, and **Dp_ma** represent delegation of permission relation where the permission type is access and manage respectively. Similarly, **Tp_a**, and **Tp_ma** represent trust of permission relation where the permission type is access and manage. Services that are considered assets are labeled with the security property that should be satisfied and their sensitivity level.

Table 3.14: ASP Formalization for SI* Model

```
role(patient)
role(hospital)
goal(provide_ medical_ service)
goal(monitor_ patient)
goal(diagnose)
.....
resource(monitoring_ data)
resource(patient_ data)
resource(prescription)
resource(computer)
resource(smart_ t_ shirt)
.....
means_ end(diagnose,prescription)
resource(smart_ t_ shirt)
del_ perm_ manage(smart_ t_ shirt,patient)
del_ perm_ manage(hospital,smart_ t_ shirt)
own(hospital,smart_ t_ shirt)
del_ perm_ manage(patient_ data,hospital)
own(patient,patient_ data)
role(pharmacy)
.....
own(pharmacy,pcomputer)
del_ perm_ manage(pcomputer,drug_ manager)
.....
del_ perm_ access(prescription,pharmacy)
own(patient,prescription)
del_ perm_ access(pharmacy,prescription)
.....
agent(kate)
agent(bob)
play(kate,patient)
play(bob,patient)
```

set of axioms that take into account if a role is the owner of a resource and the relations between resources: *stored_ in*, *part_ of* and *require*. The axioms assume the owner of a resource has the highest permission on a resource (i.e., *manage*) or that a role with the *manage* permission on a resource can delegate any permission type on the resource to another actor. In addition, if a role has a *manage* permission on an resource which stores another resource, s/he then has the *manage* permission also on the stored resource. Last, if a role has a permission on a resource, then s/he has the same permission on each subpart of the resource. For a complete list of the axioms, we refer the reader to (6). In the example, the Patient has delegated the access permission to the Pharmacy on Prescription, and thus the Pharmacy has the access permission on

Table 3.15: ASP Rules for SI* Model Instantiation

Instantiating Assets	
$A1$	$sec\_req\_instance(service\_instance(S,A,P),SP,A,P) \leftarrow sec\_req(S,SP,P) \wedge service\_instance(S,A,P) \wedge instance(A,P)$
$A2$	$sensitivity\_instance(service\_instance(S,A,P),SL,A,P) \leftarrow service\_instance(S,A,P) \wedge instance(A,P) \wedge sensitivity(S,SL,P)$
$A3$	$asset\_instance(service\_instance(S,A,P),A,P) \leftarrow sec\_req\_instance(service\_instance(S,A,P),SP,A,P) \wedge$ $sensitivity\_instance(service\_instance(S,A,P),SL,A,P)$
Instantiating Permissions	
$A4$	$permission\_instance(A,service\_instance(S,A,P) \leftarrow permission(P,S,PT) \wedge instance(A,P) \wedge service\_instance(S,A,P)$

$A1$ states that if a security property holds for a service at organizational level, this property should hold for each instance of that service. $A2$ associates a sensitivity level to an asset instance: the asset instance has the same sensitivity of the asset at organizational level. $A3$ determines if a service instance is an asset: a service instance is an asset if there is a security property that holds for the service instance and the service instance has sensitivity level. $A4$ states that an agent playing a role inherits the permissions that the role is granted on assets.

Prescription. Moreover, the Pharmacy has the manage permission on PComputer and thus it has also the manage permission on Drug Info and Prescription that are stored in PComputer. The Pharmacist and the Drug Manager are granted by Pharmacy the *manage* permission on the PComputer. In addition, the Pharmacist also gains access permission on the Prescription from the Pharmacy. Since the Drug Manager has manage permission on the PComputer and Prescription is stored in PComputer, Drug Manager has manage permission on Prescription.

The next step instantiates the SI* organizational model. We only report the axioms to instantiate the elements of the SI* model that are relevant for the insider threat identification. A complete list of the ASP rules to instantiate an SI* model can be found in (141). In the following, we introduce the rules to instantiate assets, agents' permissions on assets and the trust of permission relations between agents.

Instantiate Assets Each instance of an asset is identified with its sensitivity level. The identification is based on the rules given in Table 3.15.

In the example Prescription is an asset owned by the role Patient. The Patient role is played by the agents Bob, Kate, Jane, thus each of them owns one of the following instances of Prescription:

- *asset_ instance(service_ instance(Prescription,Bob,Patient), Bob, Patient)*,
- *asset_ instance(service_ instance(Prescription,Kate,Patient), Kate, Patient)*,
- *asset_ instance(service_ instance(Prescription,Jane,Patient), Jane, Patient)*.

Instantiate Permissions on Assets This step identifies the permissions that agents have on assets. The Pharmacy role delegates the manage permission on PComputer to role Drug Manager. Since the Pharmacy is played by the agent Pharmacy San Raffaele, and the Drug Manager is played by agents Ellen and Mary, Ellen and Mary are granted the manage permission on the instance of PComputer owned by Pharmacy San Raffaele.

Instantiate Trust of Permissions relation In this step the trust of permission relationship between agents owning assets and agents having permissions on their assets are identified. This entails determining the level of trust that the owner places in the other agent for the granted permission: the trust value can be already given or it can be computed based on trust paths by the trust model proposed in Section 3.2.3. For example, let us suppose that the agent Bob (playing the Patient role) wants to determine the level of trust with which he can grant the access permission on its asset Prescription to Ellen (playing the Drug Manager role). Since Bob has no direct trust relationship with Ellen we need to evaluate the trust value that Bob places in Ellen based on the following trust chain: *trust_ perm_ instance(Bob, Pharmacy Saint Claire, asset_ instance(service_ instance(Prescription,Bob,Patient), Bob, Patient), access, very good)* ; *trust_ perm_ instance(Pharmacy Saint Claire, Ellen,asset_ instance (service_ instance (Prescription,Bob,Patient), Bob, Patient), access, good).* Note that Ellen has access to the instance of Prescription owned by Bob because it is stored in the instance of PComputer owned by Pharmacy Saint Claire on which Ellen has been granted manage permission with good trust level and having the manage implies the access permission. Let us assume that the trust scale and the trust evaluation function are defined as follows:

- Very Good → 1
- Good → 0.8
- Neutral → 0.6
- Bad → 0.4
- Very Bad → 0.2

This means that values in the range $[0, 0.2]$ are assigned the label *Very Bad*, the range $(0.2, 0.4]$ is assigned *Bad*, $(0.4, 0.6]$ maps to *Neutral*, $(0.6, 0.8]$ is considered *Good*, and $(0.8, 1]$ denotes *Very Good*.

Let us also assume that in order to compute the trust value that Bob can place in Ellen for the access permission we use the product as the concatenator operator. Thus, the trust level for Ellen is 1 * 0.8 = 0.8 which corresponds to label *Good*. Thus, we can add to the SI* model formalization the following trust of permission relationship between Bob and Ellen: *trust_ perm_ instance (Bob, Ellen, asset_ instance (service_ instance(Prescription,Bob,Patient), Bob, Patient), access, good)*.

For the example, we are interested in determining all the possible insiders for the instance of Prescription asset owned by the Patient Bob. The reasoning supported by all the previous formalization steps report the following insiders:

- threat(Dr Stefano,asset_ instance(service_ instance(Prescription, Bob, Patient), Bob, Patient), confidentiality, moderate)
- threat(Dr Alex,asset_ instance(service_ instance(Prescription, Bob, Patient), Bob, Patient), confidentiality, moderate)
- threat(Ellen,asset_ instance(service_ instance(Prescription, Bob, Patient), Bob, Patient), confidentiality, moderate)
- threat(Mary,asset_ instance(service_ instance(Prescription, Bob, Patient), Bob, Patient), confidentiality, high)
- threat(Ellen,asset_ instance(service_ instance(Prescription, Bob, Patient), Bob, Patient), availability, moderate) threat(Mary, asset_ instance(service_ instance (Prescription, Bob, Patient), Bob, Patient), availability, high)

Dr Stefano and Dr Alex are two insiders who represent a moderate threat to the confidentiality of Prescription instance owned by Bob because they have been granted access permission on the asset instance and they are trusted good for such permission by Bob. Ellen and Mary are insiders to both the confidentiality and the availability of Prescription asset owned by Bob because the following conditions hold:

- the asset instance is stored in the instance of PComputer owned by the Pharmacy Saint Claire and Pharmacy San Raffaele

- Pharmacy Saint Claire trusts good Ellen with the manage permission on the instance of PComputer owned by the Pharmacy San Raffaele
- Pharmacy San Raffaele trusts bad Mary with the manage permission on the instance of PComputer owned by the Pharmacy San Raffaele
- Ellen and Mary thus have the same permission on the Prescription asset owned by the Patient Bob stored in the instances of PComputer owned by Pharmacy Saint Claire and Pharmacy San Raffaele respectively
- having the manage permission on an asset implies to have also the access permission on an asset
- Ellen and Mary are trusted Bob good and bad with the manage permission on the instance of Prescription owned by Bob
- manage permission is sufficient to violate the availability of a given asset while the access permission is sufficient to violate the confidentiality of an asset.

3.2.6 Discussion

Our framework provides security engineers with a reasoning that automatically produces a list of possible insiders for organizational assets and the risk they may represent to the organization. The reasoning determines if an agent is an insider for an asset and the risk the agent brings about, based on the sensitivity of the asset, the security property specified for it, the permission assigned to the agent on the asset, and the level of trust the asset owner places in the agent for the granted permission. Once the insider threat is identified, further measures can be taken by the organization.

There are two scenarios in which using our approach would be beneficial. The first scenario comprises an existing system onto which we want to analyse potential threats, thus following a reactive approach. In this case, we can exploit the existing information about the stakeholders (customers and employees) of the system in order to provide an accurate SI* model, including existing trust relationships. This would yield a set of threats that we can tackle by deploying security solutions on top of the system. In the second scenario, which represents the primary motivation in the context of this thesis, we apply the approach in a proactive manner when the system is in its early

stage of development. In this case, given that we still lack information (e.g. about concrete instances of the different roles), we can generate fake agents, services and initial relationships automatically, and use our approach to study potential threats that may originate from these instances. We can repeat the process several times so that we can extract the most likely potential threats and their cause, which can yield key information to specify new security requirements or to design new security solutions during the early phases of the SDLC.

We are aware that our approach has some limitations. First, the validity of the results of the reasoning strictly depends on the quality and completeness of the SI* model, which in turn depends on the level of expertise of the requirements engineer. However, this is not different in more traditional risk assessment processes. Second, the visual notation of SI* might not scale well for complex application scenarios and therefore simplifications of the notation should be explored.

We are planning to evaluate the strengths and limitations of our framework by conducting a controlled experiment where master students and professionals apply the framework to a real industrial application scenario.

3.3 Eliciting and Representing Trust and Reputation Requirements

This section deals with the representation of trust and reputation information early in the analysis stage. The idea is to provide requirements engineers with expressive ways to specify how trust integrates in the system, how it affects the context of the system, and how it is affected by such context. For this purpose, we integrate trust and reputation notions into the Problem Frames notation, which considers the context of a system as a first-class citizen.

First, we give an overview of Problem Frames and its extensions, and we describe how we can formally reason about and check the consistency of the models. We also present the application of our work to a smart grid scenario and a final discussion.

3.3.1 Problem Frames

Problem frames are a means to describe software development problems. They were proposed by Jackson (54), who described them as follows: *"A problem frame is a kind*

of pattern. It defines an intuitively identifiable problem class in terms of its context and the characteristics of its domains, interfaces and requirement.". It is described by a *frame diagram*, which consists of domains, interfaces between them, and a requirement.

In Problem Frames, a *machine* represents the software to be developed, and a *domain* is a part of the world we are interested in. Jackson distinguishes the domain types *CausalDomain*s that comply with some physical laws, *LexicalDomain*s that are data representations, and *BiddableDomain*s that are usually people. Domains are connected by interfaces that consist of shared phenomena, which may be events, operation calls, messages and the like. Shared phenomena are observable by at least two domains, but are controled by only one domain, indicated by an exclamation mark.

The task of the developer is to construct a *machine* based on the problem described via the problem frame approach that improves the behaviour of the environment where it is integrated, according to the requirements. Problem frames help analyzing the problems to be solved by showing which domains have to be considered, and what knowledge must be described and reasoned about.

Software development with problem frames proceeds as follows: first, the environment in which the machine will operate is represented by a *context diagram*. Like a frame diagram, a context diagram consists of domains and interfaces, but the diagram does not contain requirements. *Domain knowledge diagrams* focus on some particular domains of the context diagram and elicit further domain knowledge about them in terms of facts and assumptions. Then, the problem is decomposed into subproblems. Each subproblem is represented by a *problem diagram* containing its domains, phenomena, interfaces, and their relations to at least one requirement that expresses the subproblem.

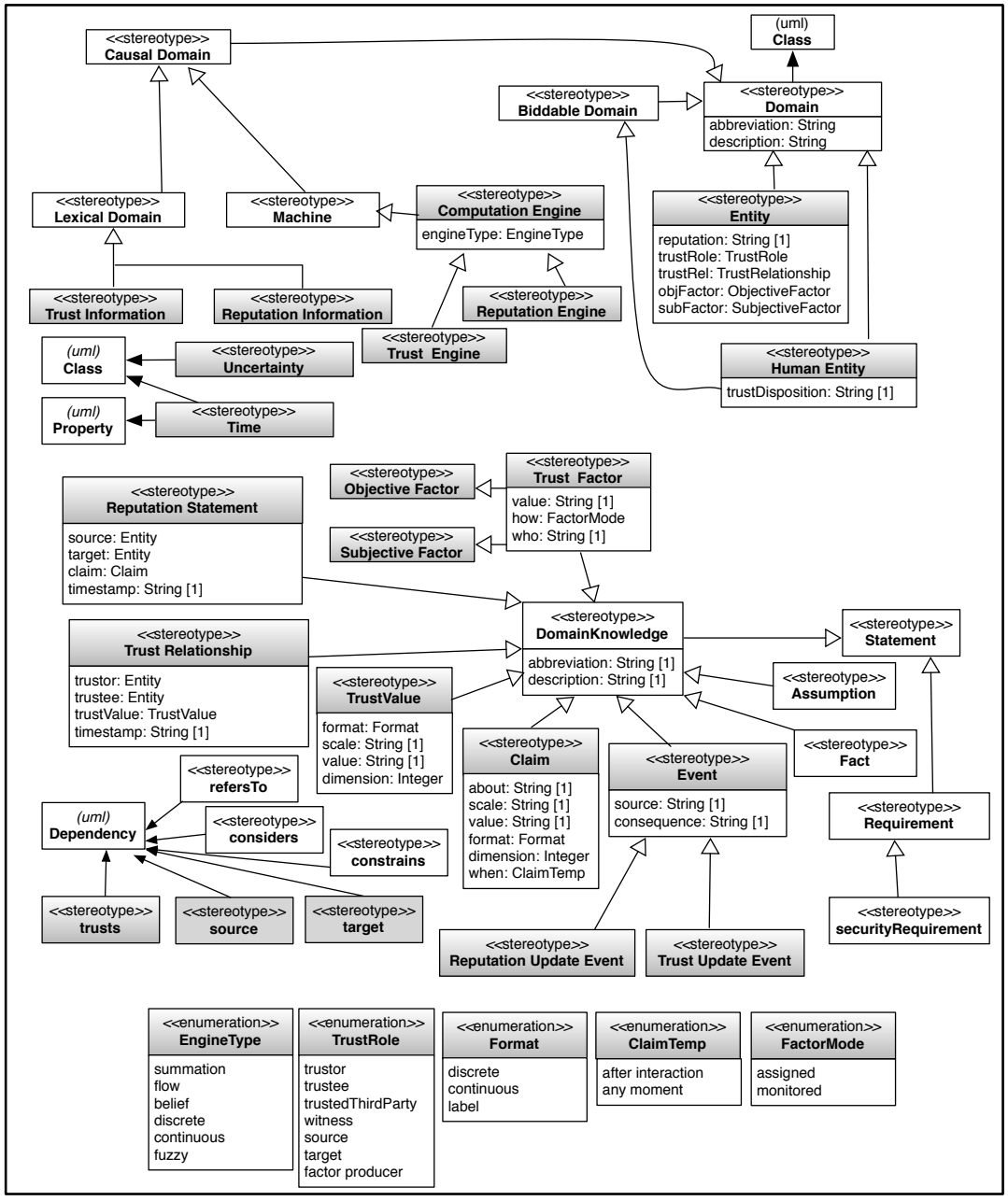
UML Profile for Problem Frames (UML4PF)[9] is a UML profile that extends class diagrams with stereotypes that contain the notions of problem frames (30, 49). The next section looks further into UML4PF and explains the extensions performed in order to add trust knowledge.

3.3.2 Trust Extensions to UML4PF

We proceed by extending UML4PF with trust and reputation concepts, as depicted in Figure 3.6.

[9] `http://www.uml4pf.org`.

Figure 3.6: A Trust Extension of the UML4PF Profile



The class with the stereotype *machine* represents the software to be developed. The classes with some domain stereotypes, e.g., *CausalDomain* or *BiddableDomain* represent *problem domains* that already exist in the application environment. The stereotype

causalDomain indicates that the corresponding domain is a causal domain, and the stereotype *biddableDomain* indicates that it is a biddable domain. Interfaces among domains are represented as associations, and the name of the associations contains the phenomena and the domains controlling the phenomena.

Domain Knowledge consists of *Statement*s about domains, in particular, *Fact*s that we can prove and *Assumption*s that we consider during software development. A *Requirement* is a specific kind of *Statement* about domains that shall hold after the *Machine* has been built. Requirements *constrain* at least one domain and can *referTo* further domains. A *securityRequirement* is a statement about the confidentiality, integrity, or availability concerns of domains and *complement* at least one functional requirement in this regard.

We use the profile to create context diagrams, domain knowledge diagrams, and problem diagrams using the elements described in Section 3.3.1. Our trust extensions for the UML4PF profile are shown in Figure 3.6[10] in grey, and are underpinned by the conceptual background provided in Chapter 2.

Trust concepts are represented as stereotypes or attributes of these stereotypes. These stereotypes are part of a domain. *Entity* is a domain and *Human Entity* is a *Biddable Domain*. *Trust Information* and *Reputation Information* are *Lexical Domain*s.

When adding trust and reputation, the goal is to build a *Machine* that encapsulates the behaviour of the trust and reputation mechanics. We specify this by stating that *Computation Engine*s are *Machine*s, which in turn can be *Trust Engine*s or *Reputation Engine*s, depending on whether they calculate trust or reputation, respectively.

Trust Engines are in charge of calculating *Trust Value*s for *Trust Relationship*s among *Entit*ies. These engines take *Trust Factor*s, associated to *Entity* as inputs, which may be *Objective Factor*s or *Subjective Factor*s. Objective factors can be assigned explicitly or can be obtained by monitoring; in any case, they are responsibility of an *Entity* playing the role *Factor Producer*. *Computation Engine*s can have different mathematical mechanics, including *belief* or *fuzzy* logics. *Uncertainty* estimates the reliability/credibility of a trust, reputation or factor value, whereas *Time* states when the *Trust Information* or *Reputation Information* was generated, or when *Factor*s were produced.

[10]Note that for readability purposes we simplified the profile and several elements are not illustrated, such as display domains and assets.

An *Entity* has reputation, plays a *TrustRole* (e.g. trustor, trustee, witness, etc), has at least one *TrustRelationship* and may have several *Objective Factor*s or *Subjective Factor*s. In addition, a *Human Entity* may have a *trust disposition* that states his propensity towards higher or lower trust values.

A *Trust Relationship* consists of a *trustor*, which is the *Entity* placing trust, a *trustee*, which is the *Entity* on which trust is placed, a *TrustValue* which is the actual value of the relationship, and a *timestamp* to keep track of the temporal factor. A trust value has a *format*, which represents the way the trust value is represented. It also has a *scale*, a *dimension* stating whether it is composed of a single *value* or a tuple of *value*s.

*Entit*ies playing the role *Source* can make *Claim*s about other *Entit*ies with role *Target*. This information is aggregated in the form of *Reputation Statement*s, which are used by *Reputation Engine*s to compute reputation scores. A *SourceEntity* can make *Claim*s *after an interaction* or just asynchronously at *any moment*.

A *Reputation Statement* consists of an *Entity* playing the *source* role, an *Entity* playing the *target* role, a *Claim* and a *timestamp* to keep a record of the temporal dimension. A *Claim* is *about* some feature of the *target* entity, it has a *scale*, a *format*, which represents the way the claim is represented, a *dimension* stating whether it consists of a single *value* or a tuple of *value*s. A *Claim* also specifies *when* the claim is issued, which can be right *after an interaction* or at *any moment*.

Finally, *Event*s are circumstances in the system that trigger a trust or reputation update. These events can be visualized by dynamic diagrams, such as sequence diagrams, and have a *source* that triggers it, and a *consequence*.

3.3.3 Formal Checking of Trust

One advantage of using UML is that it enables requirements engineers to query the models using the Object Constraint Language (OCL). This can be useful to find probable missing information. Table 3.16 lists the expressions that can be formulated in OCL for both consistency checks and reasoning support. Consistency checks identify mistakes prone to being made by requirements engineers, whereas reasoning support expressions encompass issues that may require a detailed discussion among engineers.

Listing 3.1 and 3.2 show the listings of two of the OCL expressions. The interested reader can check the work by Beckers et al. (13) for the rest of expressions.

Listing 3.1: IDHE001. List all biddable domains that are not a human entity

```
-- List all biddable domains that are not a human entity

 let stereotypeMain : String =
  'BiddableDomain'
  in

  let
  biddables : Set (Class) =

  Class.allInstances()->select(
  let first : Set(Stereotype) =
  getAppliedStereotypes()->asSet()
  in
  first->union(first->closure(
  general.oclAsType(Stereotype)))
  .name->includes(stereotypeMain))
  in

   let stereotypeHumanies : String = 'Human_Entity' in

   let
   humanies : Set (Class) =

   Class.allInstances()->select(
   let first : Set(Stereotype) =
   getAppliedStereotypes()->asSet()
   in

   first->union(first->closure(
   general.oclAsType(Stereotype)))
   .name->includes(stereotypeHumanies))
   in

  biddables - humanies
```

Listing 3.2: CCTV001. Check that all dependencies with a trust relationship have a dependency to a TrustValue

```
Check that all dependencies with a trusts relationship have a dependency to a
    TrustValue


 -- Check that all dependencies with a trusts relationship have a depen- dency
     to a TrustValue


 -- Get all dependencies with a trusts stereotype

     let
      stereotype : String = 'trusts'
```

```
11      in
12
13     let
14         Trusts : Set (Dependency) = Dependency.allInstances()->select(
15
16      let first : Set(Stereotype) = getAppliedStereotypes()->asSet()
17      in
18
19     first->union(first->closure(
20    general.oclAsType(Stereotype)))
21    .name->includes(stereotype))
22     in
23
24
25 -- Get all Trust Values
26
27
28   let stereotypeMain : String =
29    'Trust_Value'
30    in
31
32    let
33    TrustValueClasses : Set (Class) =
34
35    Class.allInstances()->select(
36    let first : Set(Stereotype) =
37    getAppliedStereotypes()->asSet()
38    in
39
40    first->union(first->closure(
41   general.oclAsType(Stereotype)))
42   .name->includes(stereotypeMain))
43    in
44
45 -- Get all Target Dependencies of trust values
46
47     let stereotypeTargetOne : String = 'trusts' in
48
49   let haveTrustValues : Set (Dependency) =  TrustValueClasses->select(
50   clientDependency.target.getAppliedStereotypes()
51   .name->includes(stereotypeTargetOne))
52   .clientDependency.target.oclAsType(Dependency)->asSet()
53 in
54
55
56 -- Subtract trust dependencies that have trust values
57
58
59 let notHaveTrustValues : Set (Dependency) =
60
61 Trusts->-(haveTrustValues)
62 in
```

```
63
64  notHaveTrustValues->isEmpty()
```

3.3.4 Application Example: Smart Grid

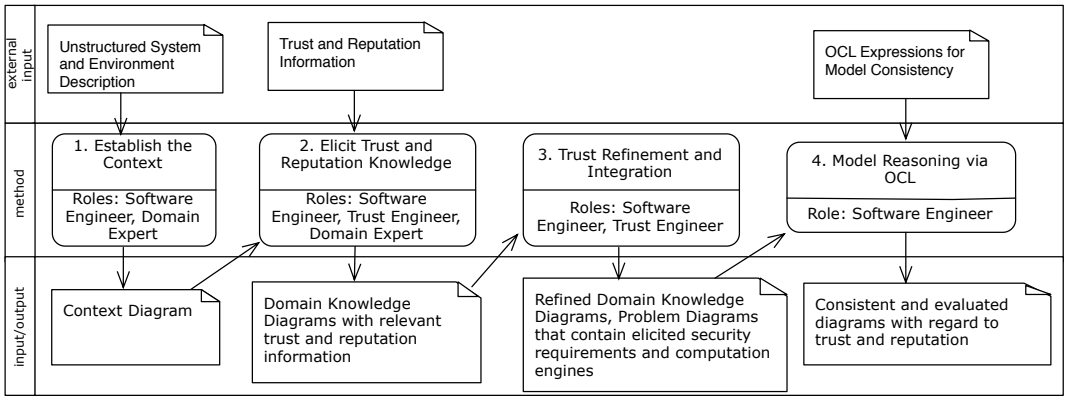
Table 3.16: OCL Expressions that support Trust-based Security Reasoning and Consistency Checks

OCL-EXPR-ID	Referenced Class	Expression	Supporting Analysis Questions
Reasoning Support			
IDHE001	HumanEntity	- List all biddable domains that are not a human entity	- Are human entities missing?
IDEN001	Entity, HumanEntity	- List all domains that are not entities or human entities	- Are some entities or human entities not elicited yet?
IDHE002	HumanEntity	- List all human entities that do not have a trusts relation.	- Are trust relations of HumenEntities missing?
IDEN002	Entity	- List all entities that do not have a trusts relation.	- Are trust relations of Entities missing?
TRTE001	TrustEngine	- List all machine domains that have a direct relation to a TrustEngine	- Are Trust Engines missing in the model?
TRRE001	Reputatio-nEngine	- List all TrustEngines that have a direct relation to a ReputationEngine	- Are ReputationEngines missing in the model?
TRJE001	TrustFactor	- Check that the how attribute is set and it is set either to "assigned" or "monitored".	- Are all the trust factors either “assigned” or “monitored”?
TRCL001	Claims	- Check that claims have set the when attribute to either to “after interaction” or “any moment”.	- Do all claims specify when they must be provided.
TROF001	ObjectiveFactor	- List all trust relationships that have no objective factors.	- Are all objective factors of the entity considered?
TRSF001	SubjectiveFactor	- List all trust relationships that have no subjective factors.	- Are all subjective factors of the entity considered?
Consistency Checks			
CCRS001	HumanEntity	- Check that all HumanEntities have the value trustRole set.	- Are HumanEntities modelled correctly ?
CCRS002	Entity	- Check that all EntitiesEntities have the value trustRole set.	- Are Entities modelled correctly ?
CCCL001	Claim	- Check that all sources of claims are a Human Entity	- Are claims modelled correctly ?
CCCL002	Claim	- Check that all targets of claims are an Entity or Human Entity	- Are claims modelled correctly with respect to entities?
CCCL003	Claim	- Check that all claims have targets and sources	- Have claims an origin and a target ?
CCSF001	SubjectiveFactor	-Have subjective factors the who value set and refer to a trust relationship?	- Are trust relationships modelled considered subjective factors ?
CCTV001	TrustValue	- Check that all dependencies with a trusts relationship have a dependency to a TrustValue	- Are trust relationships modelled completely ?
CCLTR001	Trustor, Trustee	- Check that trust relationships have a trustor and a trustee.	- Are all trust relationships modelled correctly?
CCLTR002	TrustFactor	- Check that All classes with a stereotype trust factor including the inheriting classes subjective and objective factor have a dependency to a trusts relationship or to an entity	- Are all trust factors refer to trust relationship or to entities?

As an example of our approach, we use the Common Criteria protection profile for the smart metering gateway (21), which defines security requirements for this element.

The gateway is a part of the smart grid, which is a commodity network that intelligently manages the behaviour and actions of its participants. The commodity consists of electricity, gas, water or heat supply that is distributed via a grid or network. The benefit of this network is envisioned to be a more economic, sustainable and secure supply of commodities. Smart metering systems meter the production or consumption of energy and forward the data to external entities. This data can be used for billing and steering the energy production. We use the trust and reputation extensions of UML4PF introduced in Section 3.3.2 in order to integrate trust and reputation requirements. Likewise, we propose the methodology depicted in Figure 3.7 and we explain and apply each step next.

Figure 3.7: Methodology Proposed for Engineering Trust and Reputation Requirements into Systems



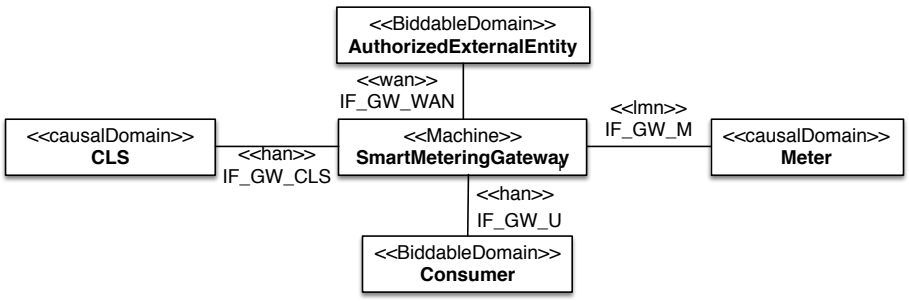
Step 1: Establish the Context Trust relationships are only valid for a specific context. The software engineer and the domain expert describe the context of the software development in a context diagram. This diagram describes the machine in its environment using domains and interfaces between them. A set of textual functional requirements refers to the domains in the context diagram. Afterwards, the trust engineer[11] elicits assets and security requirements for them.

[11]We use the term *trust engineer* to refer to both a security engineer and an expert in trust models. In some cases, these profiles may be covered by the same person. In others though, two different persons with more specialized profiles may be required.

Figure 3.8 shows the context diagram that describes the machine to be built in its environment. The *Machine* is the *SmartMeteringGateway*, which serves as a bridge between the Wide Area Network *wan* and the Home Area Network *han* of the *Consumer*. The *Meter* is connected to the machine via a Local Metrological Network *lmn*. This is an in-house equipment that can be used for energy management. The *Controllable Local System (CLS)* is a device located in the consumer house and which is connected to the smart grid system; therefore, the energy of the CLS can be controlled by the system. Some examples include the heater, the oven or the lights over an area of the house.

As for requirements, the *Meter* sends meter data to the *SmartMeteringGateway*, which can store this data. The *Meter* can also receive updates from the *AuthorizedExternalEntity* forwarded via the *SmartMeteringGateway*. The *AuthorizedExternalEntity* receives meter data in fixed intervals from the *SmartMeteringGateway*. The *Consumer* can retrieve meter data from the *SmartMeteringGateway*. The *Consumer* can also configure the *SmartMeteringGateway*, send commands to the *CLS*, receive status messages from the *SmartMeteringGateway* and store user data in it.

Figure 3.8: Context Diagram for the Smart Metering Gateway



Step 2: Elicit Trust and Reputation Knowledge The domain expert and the trust engineer have to work together. The former elicits trust-unaware domain knowledge diagrams, whereas the latter (together with the former) provides an initial *trust domain knowledge*, where the high level aspects of the trust and reputation models are first sketched. These aspects include specifying trust entities, their trust relationships, claims, and trust factors.

Once the context is established, trust and reputation information must be elicited. We show in Figure 3.9 a domain knowledge diagram focusing on the main elements of one trust relationship between the *HumanEntity Consumer* and the *Entity CLS*. The trust relationship has a *TrustValue* and there is a *SubjectiveFactor* associated to the *Consumer*.

On the other hand, Figure 3.10 shows relevant information for reputation purposes. Concretely, we are specifying which entities can make *Claim*s about others, and which objective factors are considered to yield those claims. In this example, a *HumanEntity AuthorizedExternalEntity* can make *Claim*s about the *Entity CLS*, and the *Objective-Factor UnplannedReparis* refers to the *CLS*.

Figure 3.9: Domain Knowledge Diagram - Analysing the Trust Relationship Consumer-CLS

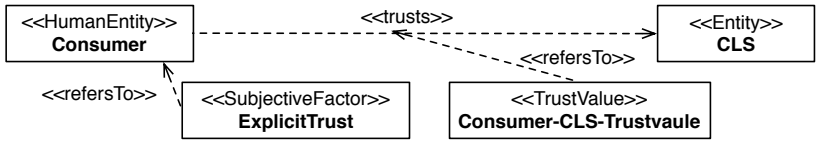
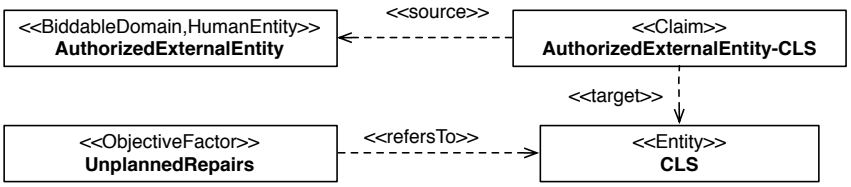


Figure 3.10: Domain Knowledge Diagram - Analysing Reputation Information



Step 3: Trust Refinement and Integration The information in the trust domain knowledge diagrams is refined in this step by the software and trust engineers. The final diagrams contain detailed information about trust and reputation relationship, e.g. roles played by the entities, insight on the claims, the trust values, and objective and subjective factors. Figure 3.11 shows the refinement of the previous domain knowledge diagrams.

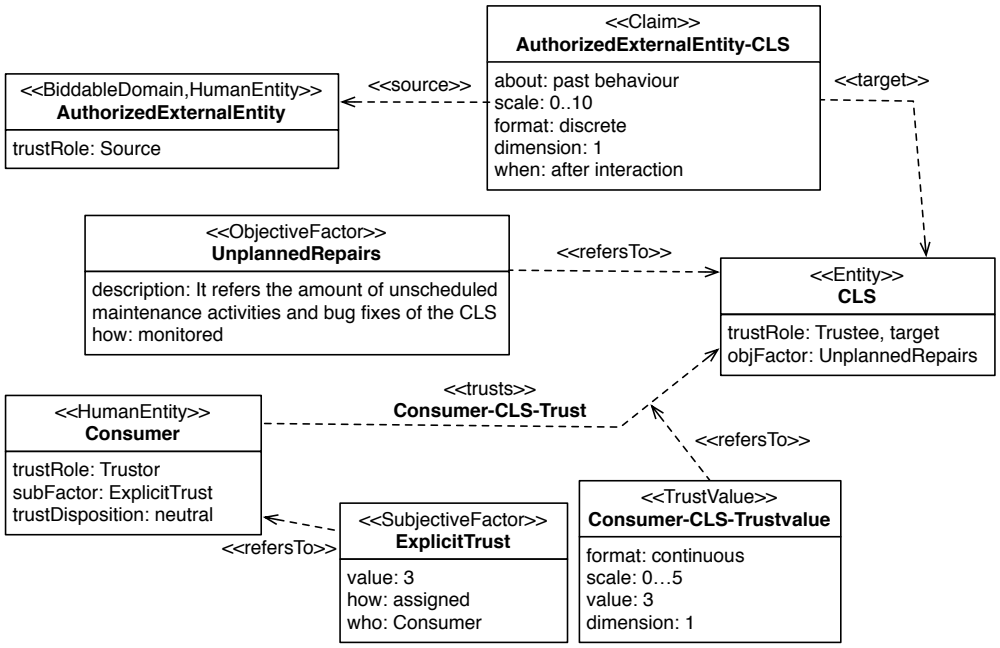
The Consumer plays a *trustor* role in the *Consumer-CLS-Trust relationship*, it uses the subjective factor *ExplicitTrust* for this relationship and his trust disposition is neu-

tral[12]. The *ExplicitTrust* subjective factor has 3 as initial value and is *assigned* by the Consumer. The *Consumer-CLS-Trustvalue* is a unidimensional continuous value between 0 and 5, with threshold value 3. This latter value refers to the threshold over which we assume that a trustor trusts a trustee. The CLS plays a *trustee* role in the *Consumer-CLS-Trust* relationship, although it also plays the *target* role with regard to the *AuthorizedExternalEntity-CLS* claim. It presents an objective factor, *UnplannedRepairs*, which is *monitored* (in contrast to manually assigned). The *AuthorizedExternalEntity* plays a *source* role because it can make claims about the CLS *after an interaction* with it. Claims are about the past behaviour of the target, and are represented by a unidimensional discrete number between 0 and 10.

In addition to refining trust and reputation information, the trust engineer and the software engineer collaborate to analyse how the respective trust and reputation engines integrate into the system-to-be and their relationship with the system requirements and the machine. Figure 3.12 depicts the interactions between the three ma-

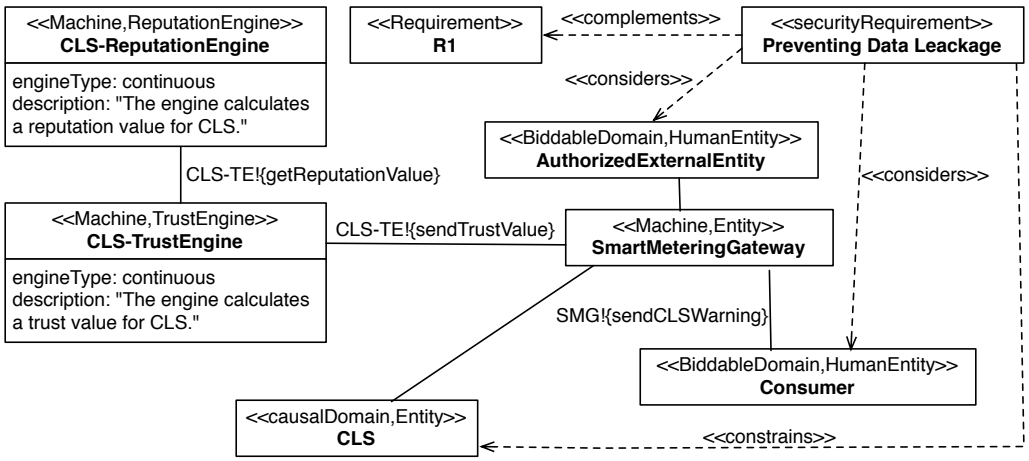
[12]We consider that for this scenario, a neutral trust disposition is reasonable, whereas other scenarios might require assuming that trust dispositions are lower or higher.

Figure 3.11: Domain Knowledge Diagram - Refinement of Trust and Reputation Information



chines of the system: the *SmartMeeteringGateway*, the *CLS-TrustEngine* and the *CLS-ReputationEngine*. Both computation engines yield continuous values. The trust engine can retrieve reputation values from the reputation engine in order to compute trust values[13]. The *SmartMeeteringGateway* can retrieve trust information and act accordingly.

Figure 3.12: Problem Diagram - Describing Trust and Reputation Engines

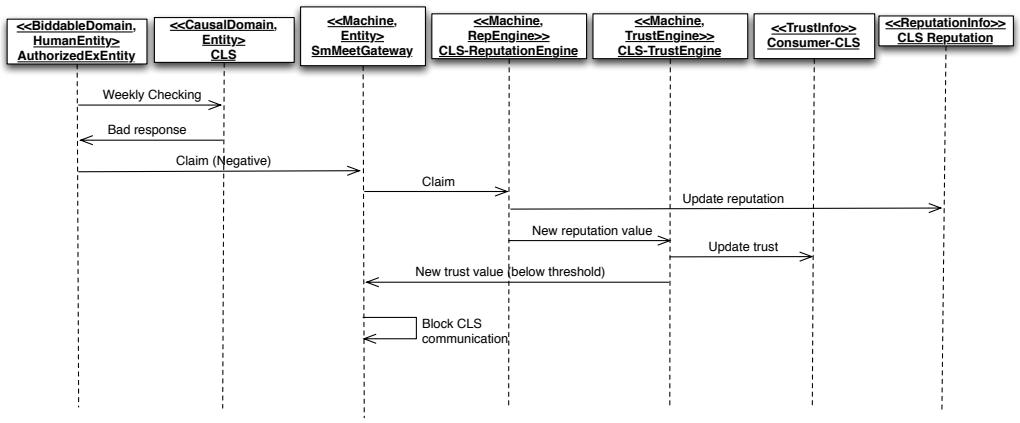


We consider the following functional requirement of the smart metering gateway in our example: **R1** *The CLS can receive energy consumption data from the Meter.* We elicit the security requirement *Prevent Data Leakage* that *complements* the functional requirement **R1**. If the value of the *Consumer-CLS-Trustvalue*, which is computed by the trust engine, is above the minimum trust threshold (initially set to 3), no action shall be taken. Otherwise, communications with the CLS should be blocked by the *SmartMeeteringGateway*. In addition, the claims issued by the *AuthorizedExternalEntity* (i.e. *AuthorizedExternalEntity-CLS*) are used by the reputation engine to yield a reputation value, which is fed into the trust engine. Both the trust events and the consequences of trust decisions can be sketched by means of dynamic diagrams, as depicted in the example in Figure 3.13.

Step 4: Model Reasoning via OCL We use the OCL expressions in Table 3.16 in this step. In particular, we illustrate the expression CCCL001 in detail in the following.

[13]This is the traditional way of relating trust and reputation: reputation is a valuable source of information for trust computation.

Figure 3.13: Sequence Diagram - Describing Trust and Reputation Events



Everything starts with the *AuthorizedExternalEntity* performing a weekly check on an CLS. After such interaction (which we assume is negative), the *AuthorizedExternalEntity* issues a negative claim, which is forwarded by the *SmartMeeteringGateway* to the *ReputationEngine*, which uses this claim to recompute a new reputation value. This new reputation value is sent to the *TrustEngine*, which recalculates the trust value between the *Consumer* and the *CLS*, and this new trust value (together with some other information such as the the threshold of the relationship) is sent back to the *SmartMeeteringGateway*. As a result of the new trust value being lower than the threshold, the SmartMeeteringGateway blocks the communication with the CLS, because the *Consumer* no longer trusts the CLS.

The expression shown in Listing 3.3 collects all classes with the stereotype *Claim* (lines 1-8) and all dependencies with the stereotype *source* (lines 9-15). The expression filters the dependencies that start at a class with the stereotype *Claim* and end at a class with the stereotype *HumanEntity* (lines 16-21). Finally, the expression subtracts the classes with the stereotype *Claim* that are at the end of the previously mentioned dependencies from all classes with the stereotype *Claim* (lines 22-24). In our case all the claims originate from the human entities and the expression returns an empty set. Otherwise we would get a list of classes for analysis.

Listing 3.3: CCL001. List all sources of claims that are not a Human Entity

```
1 let stereotypeMain : String = 'Claim' in
2 let
3   claimClasses : Set (Class) =
4   Class.allInstances()->select(
5   let first : Set(Stereotype) = getAppliedStereotypes()->asSet() in
6   first->union(first->closure(general.oclAsType(
7   Stereotype))).name->includes(stereotypeMain))
```

```
8  in
9  let stereotype : String = 'source' in
10 let
11    Sources : Set (Dependency) = Dependency.allInstances()->select(
12    let first : Set(Stereotype) = getAppliedStereotypes()->asSet() in
13    first->union(first->closure(general.oclAsType(
14    Stereotype))).name->includes(stereotype))
15 in
16 let stereotypeSource : String = 'Claim' in
17 let stereotypeTarget : String = 'Human_Entity' in
18 let
19 DependencyClaims : Set (Dependency) =
20  Sources->select(source.getAppliedStereotypes().name ->includes(
       stereotypeSource) and target.getAppliedStereotypes().name ->includes(
       stereotypeTarget))
21 in
22 let correctClaims : Set (Class) =
23  DependencyClaims.source.oclAsType(Class)->asSet() in
24  claimClasses-correctClaims
```

We illustrate our tool support in Figure 3.14, which shows the modelling of a UML4PF trust model. Figure 3.15 depicts an OCL expression executed on that model and its results.

3.3.5 Discussion

The proposed methodology uses an extension over the problem frames notation in order to accommodate trust and reputation concepts and relationships among these concepts. Intensive context-awareness is an envisioned property of future, complex software systems, and problem frames fit well due to their focus on describing the context around the system-to-be. Also, the context becomes of paramount importance when analysing trust relationships and reputation information, because most of the valuable sources of information for computing trust and reputation will come from this context.

We discussed the approach with the security practitioners in the ClouDAT project[14] in a brainstorming session. We presented the methodology to practitioners in the field of security engineering that were familiar with the Protection Profile. As a result, we found that our methodology helped the practitioners distinguish between the concepts of trust and reputation.

The practitioners mentioned that this structured procedure helps identify trust relationships, supports the identification of reputation claims, helps not forget relevant

[14] http://ti.uni-due.de/ti/clouddat/en/

entities and their attributes, and supports the creation of consistent trust and reputation diagrams.

However, the following concerns towards our methodology were raised:

- The results of the OCL reasoning expression might lead engineers to add random elements to achieve completeness.
- Reading the output of all expressions might be too time consuming.
- The UML profile and the methodology have to be learned beforehand.
- Our method does not integrate into common security development life cycles such as the Microsoft SDL[15].

The outcome of our methodology is a set of requirement artifacts that represent functional requirements and trust concerns of the system. Given that these artifacts are shaped around the problem frames approach, and that this approach encourages the modularization of the system into domains, the artifacts provide a good starting point for sketching the architecture of the system. In our methodology, trust and reputation models are decomposed in their constituent elements, which provide developers with sufficient information to implement the models and to integrate them into the system in the next stages of the development life cycle.

[15]Microsoft Security Lifecycle `http://www.microsoft.com/security/sdl/default.aspx`.

Figure 3.14: Tool-Supported Modelling of Trust-Aware UML4PF

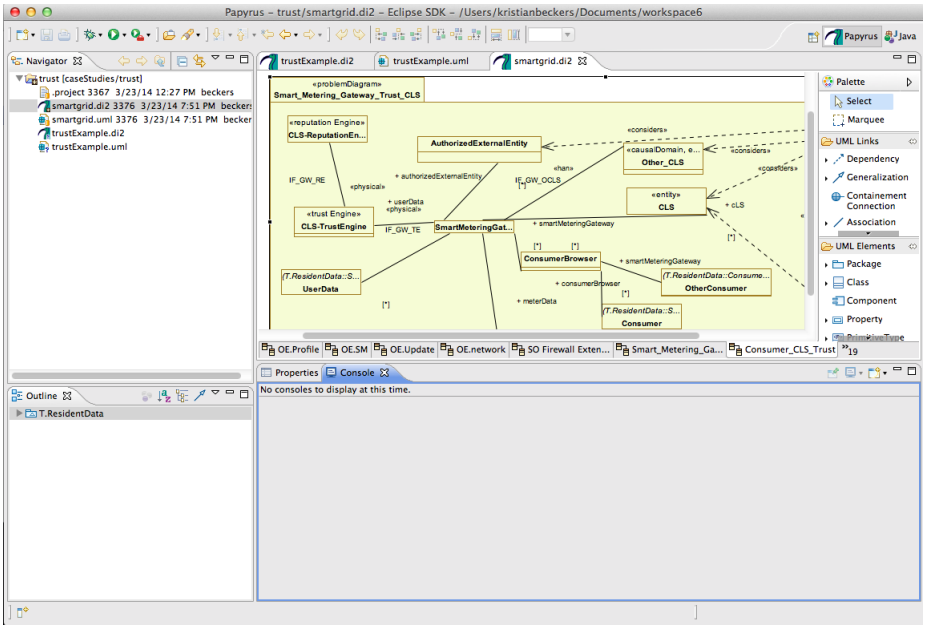
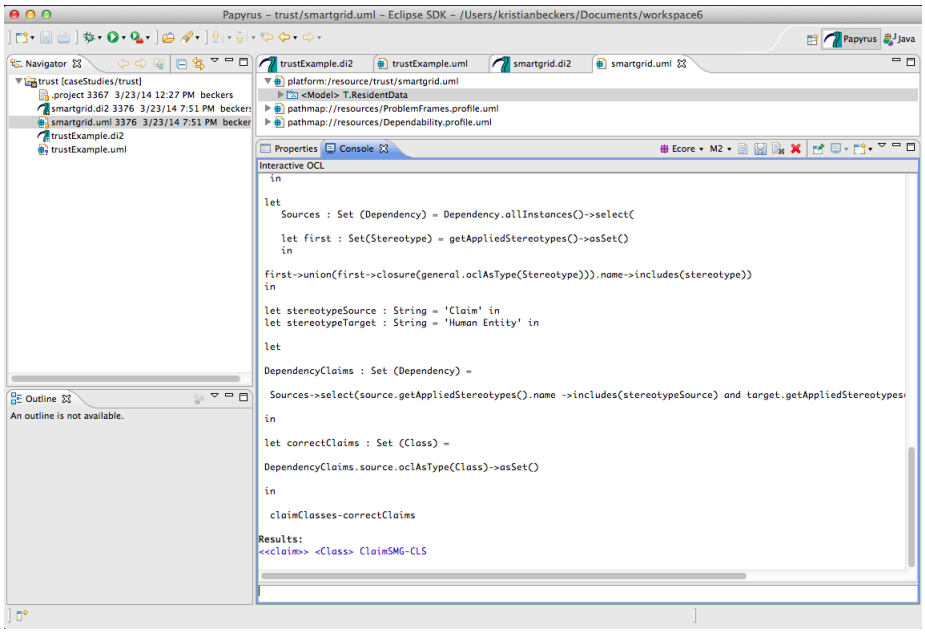


Figure 3.15: Tool-Supported OCL Evaluation



3.4 Designing Trust and Reputation Solutions

We provide an extension over UML called UMLTRep in order to help requirements engineers and software designers have a clear understanding of the trust and reputation requirements of the system, as well as a means to provide an initial specification of the trust and reputation solutions. We choose UML because it is a *de facto* standard in the industry and because other relevant security-oriented profiles exist that could be potentially integrated with ours.

This approach can be considered complementary to the one presented in Section 3.3, where we extended the Problem Frames notation and methodology to specify and integrate trust and reputation requirements into the system. The main difference with the previous approach is that now we are not so concerned with the context where the system will operate. Our focus now is on providing a description of trust and reputation elements that allows an easy implementation in the subsequent phases of the SDLC. We also provide now two valuable pieces of information that were missing in the Problem Frames approach: the rationale behind using trust and reputation, that is, the decision-making process that trust and reputation aim to assist; and an infrastructure view of the system through deployment diagrams. This, together with the rest of information conveyed by class diagrams, provide a holistic view of the trust and reputation models that we want to implement.

Even when we do not extend any behavioural diagram, we consider indispensable their use (e.g. activity diagrams) in order to show the possible trust events that can be triggered in the application. We consider that a trust event is something that may occur in the system so that it triggers an update of trust relationships or reputation. These diagrams are useful to represent the dynamic aspects of trust and reputation models, but more importantly, to make clear the glue and interaction patterns between the trust and business layers of the application. There are three important questions that these diagrams should answer: who, how and what? *Who* refers to which actor in the system can trigger the event, whereas *how* indicates how the actor actually triggers the event. The consequences of the event is the answer to *what*. In Section 3.4.4 we discuss how this information can be presented in an activity diagram.

The following sections describe the extended diagrams in further detail and explain how the profile can be used in an eHealth case study, as well as some lines for future

research.

3.4.1 Use Case Diagram

The goal of use case diagrams in the context of trust is to depict, at a glimpse, the trust relationships that exist between the different entities in the system. We can also describe which entities can make a claim about which other entities, thus incorporating reputation information in the diagram. There is however more interesting information that we can represent in this diagram: the rationale for the trust or reputation model and how it affects to the use cases. The extensions performed on the use case diagram are summarized in Table 3.17.

Table 3.17: Use Case Diagram Extensions

Stereotype	Base Class	Explanation
Trustor	Actor	Entity playing the trustor role
Trustee	Actor	Entity playing the trustee role
Witness	Actor	Entity playing the witness role
Source	Actor	Entity capable of making a claim
Target	Actor	Entity capable of receiving a claim
FactorProducer	Actor	Entity capable of producing a factor
Trusts	Connector	Trust relationship
Claims	Connector	Source makes a claim about a Target
Decides	Connector	Use case affected by a trust/reputation decision

Trustor, *trustee*, *witness*, *source*, *target* and *FactorProducer* are roles that entities can play in the system. Trust relationships are made explicit by means of the extension *trusts*, whereas *claims* represent that a given *source* can make a claim about a given *target*. As the ultimate goal of trust is aiding in making a decision, we also add the *decides* connector, which captures the idea that a use case can be affected by trust or reputation information. An entity could perform the same use case in different ways (or even could decide not to perform it at all), and this decision can be influenced by trust or reputation information.

In addition to the previous UML extensions, we define two adornments: *decision criteria* and *context*. The former is used to annotate the *decides* relationship between

an entity and a use case, and it specifies whether the decision is based on trust or reputation. The latter annotates *trusts* and *claims* relationships and specifies their context. This captures the idea that trust and reputation are context dependent.

3.4.2 Class Diagram

Class diagrams can provide more insight about certain aspects of trust and reputation. The stereotypes used to extend class diagrams are depicted in Table 3.18. We find the same stereotypes as in the use case diagram extension regarding the roles of the entities. Also, we find *TrustRelationship*, which represent the trust relationship between a *trustor* and a *trustee*, and *Claim*, which captures the notion of a claim made by a *source* entity about a *target* entity. We add also three important notions for the evaluation of trust and reputation, namely *TrustEngine*, *ReputationEngine* and *Factor*. They represent how trust and reputation are computed, and the factors considered for such computation.

Table 3.18: Class Diagram Extensions

Stereotype	Base Class	Explanation
Trustor	Class	Entity playing the trustor role
Trustee	Class	Entity playing the trustee role
Witness	Class	Entity playing the witness role
Source	Class	Entity capable of making a claim
Target	Class	Entity capable of receiving a claim
FactorProducer	Class	Entity capable of producing a factor
TrustRelationship	Class	Trust relationship between trustor and trustee
Claim	Class	Claim that a source makes about a target
TrustEngine	Class	Engine in charge of updating a trust relationship
ReputationEngine	Class	Engine in charge of computing a target's reputation
Factor	Class	Factor used by a trust or reputation engine

Tagged values are used in order to define more precisely the aforementioned concepts. The list of tagged values is shown in Table 3.19. To mention some of them, *subjective factors* and *objective factors* refer to subjective and objective factors of trustors and trustees. *Dimension* is the number of components of a trust or reputation value, and

how specifies whether the value of a factor is explicitly assigned (interactively) by the entity playing the role *factorProducer* or is monitored by another system.

Table 3.19: Tagged Values for Class Diagrams

Value	Class	Explanation
type	Trustor, Trustee, Witness, Source, Target	The type of entity (i.e. human, system)
subFactor	Trustor, trustee	Subjective factors
objFactor	Trustor, trustee	Objective factors
context	TrustRelationship, Claim	Context
dimension	TrustRelationship, Claim	Dimension of a trust relationship or a claim
scale	TrustRelationship, Claim, Factor	Upper and lower bounds
default	TrustRelationship	Default/Bootstrapping value
format	TrustRelationship, Claim	Quantitative vs. qualitative
display	ReputationEngine	Visualization by human entities
engine	Engine	Type of computation engine
factors	Engine	List of factors used by the engine
attribute	Factor	Attribute(s) captured by the factor
source	Factor	System or entity that triggers the factor update
how	Factor	Assigned vs. monitored

Note that some of these tagged values could be almost directly mapped to attributes of design classes, whether others are just informative and require further refinement. For example, *attribute* represents the attribute(s) captured by a factor. This information might be useful for aiding designers to keep in mind what the factor actually should represent, the semantics of the factor, but could hardly be mapped directly to the attribute of a design class.

3.4.3 Deployment Diagram

Deployment diagrams are useful as they represent the system from the infrastructure point of view, and trust and reputation must often be considered not only at the application level (i.e. trust among entities or among system components), but also at the infrastructure level (104). Platforms and networks can trust each other and they can even hold reputation values. This is particularly useful when designing large-scale distributed systems, where a given processing node (e.g. a mobile phone or a server) can choose among different nodes in order to collaborate or communicate information.

How trust and reputation information is derived from lower to higher level abstraction (e.g. from a system component to a system package, or from a system package to

a processing node) is an interesting field of research but it is out of the scope of this section. However, we must be consistent. For example, if we specify that a node decides to use another node due to its reputation, we must consider, in the deployment diagram itself, where this reputation is going to be stored. This in turn may drive the design of a new claim that the first platform (source) can make about the second one (target), which can be further detailed in a class diagram, leading to a new design iteration.

The extensions performed on deployment diagrams are shown in Table 3.20. We can specify which node acts as reputation manager in a centralized reputation model. Reputation managers compute reputation, store it, and distribute it (or just publish it) when necessary. The *decides* stereotype captures the decision process made by one entity (processing node) when communicating with other processing nodes. As in the case of use case diagrams, this stereotype can be adorned in order to make explicit whether this decision is based on trust or reputation with *decision criteria*. Finally, we also add a tagged value *entities* to specify the reputation of which entities the reputation manager should store.

Table 3.20: Deployment Diagram Extensions

Stereotype	Base Class	Explanation
ReputationManager	Node	Node that acts as reputation manager
decides	Connector	Trust-based decision

The next section puts all the concepts discussed in this section together by applying them to an eHealth scenario.

3.4.4 Application Example: eHealth

We present in this section how we can apply UMLTrep to a real scenario. The case study comes from the NESSoS project[16] and belongs to an eHealth scenario as described in a project deliverable (87).

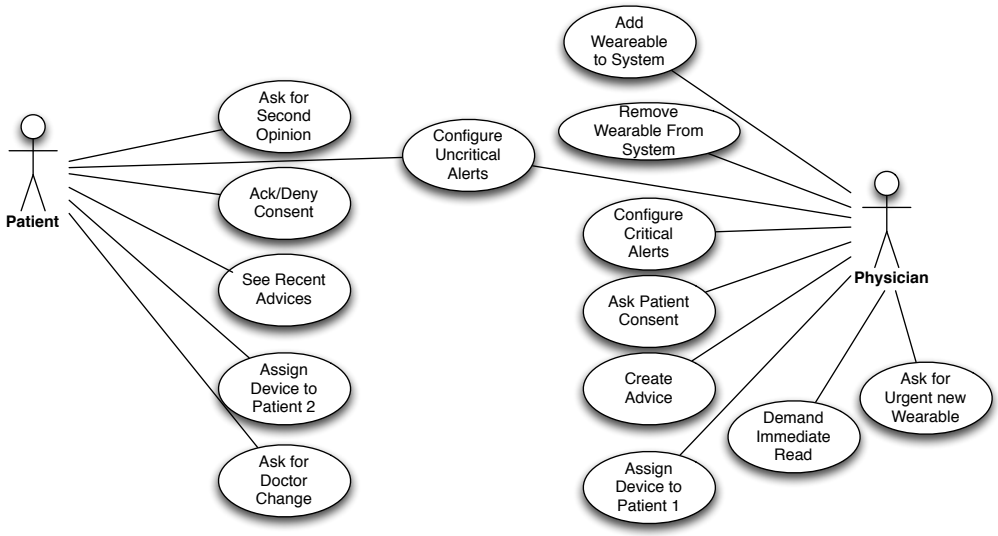
The case study presents a patient monitoring scenario, which aims to collect health-related data independently of the location of the patient. This is useful for patients, who can receive immediate feedback under critical situations and be assisted by physicians at any moment and place. In order to make this scenario feasible, the patient must

[16]http://www.nessos-project.eu

wear a device capable of measuring vital signs (e.g. blood pressure). This device must be able to send this information to other systems that will show it to physicians for monitoring purposes.

The goal is to build a web application through which the physician and the patient can interact in a trusted way[17]. In this application, the physician can add and remove a wearable device to the system, start the process to assign the device to a patient, configure both critical and uncritical alerts, ask patient consent to use his data for research purposes, create an advice for the patient based on the patient's data, demand an immediate reading from the wearable and start the process to change a patient's wearable. Patients can configure uncritical alerts, ask for second opinions (to other physicians), accept or deny consent about their data being used for research, read the physician's advices, complete the device assignment process started by the physician and demand a physician change. These requirements are represented by the use cases illustrated in Figure 3.16.

Figure 3.16: Use Case Diagram



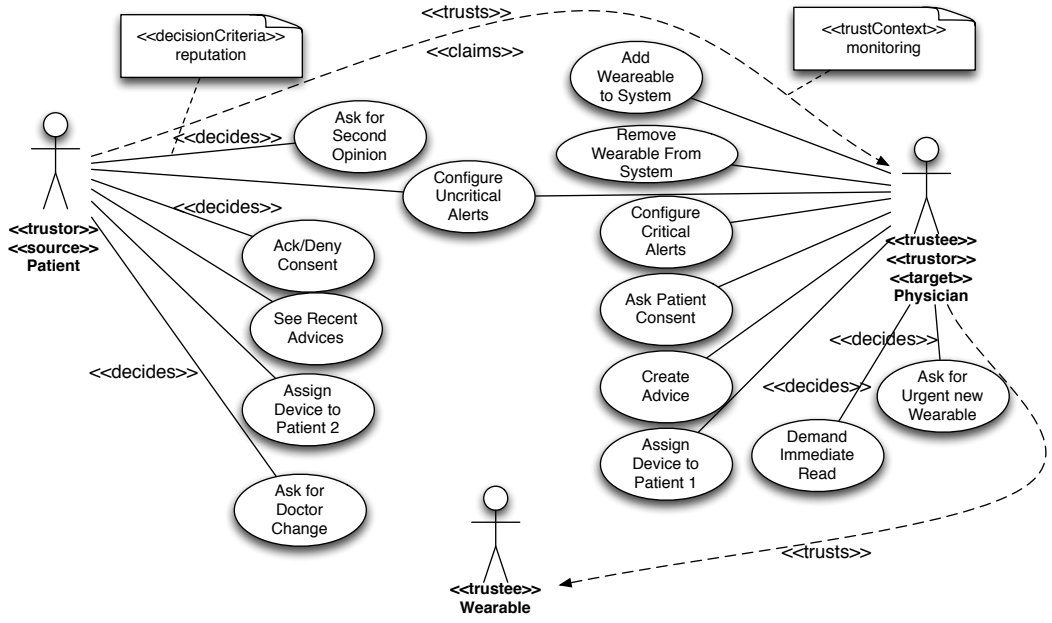
Ensuring security in this scenario is very important. On the one hand, it is required to ensure that data of one patient do not appear in the EHRs of other patients. Confidentiality and integrity of the data, as well as integrity of the wearable is also required.

[17] As discussed in Section 1.1.3, we assume that basic security mechanisms (e.g. TLS/SSL for Internet communications) are underpinning the trust and reputation solutions that we will develop.

Yet even though there are important hard security requirements, the system must also be trust-aware, in the sense that physicians and patients must trust the information provided by each other.

A possible trust-aware use case diagram is shown in Figure 3.17. We state that there is a trust relationship between the patient and the physician. The patient plays a *trustor* role and the physician plays a *trustee* role. In addition, there is a *trusts* connector, which is adorned by the *context* where this trust relationship is set, namely *monitoring*. There is another trust relationship between the physician (who therefore also plays a *trustor* role) and the wearable. The patient also plays the *source* role and can therefore make claims (*claims* connector) about the physician, who plays in this case the *target* role.

Figure 3.17: Trust-aware Use Case Diagram



Up to now, we have defined the main entities, the trust roles they can play, and the trust relationships and possible claims that the application considers. We also need to include for which purpose this information is going to be used, and this is the role of the *decides* connector. The patient may decide to ask another physician for second opinion. In order to decide who this other physician is, he uses reputation information about the physician (annotation *decision criteria*). Also, the physician may ask for a

new wearable if his trust in the actual wearable falls below a certain threshold. Thus, we are using trust and reputation to help entities make decisions at runtime.

Claims and trust relationships can be further refined in trust-aware class diagrams, as shown in Figure 3.18, 3.19 and 3.20. Regarding the patient-physician relationship, we specify the context of this relationship, which should be consistent with the context in the use case diagram, the dimension and format, which are 1 and numeric in this case, the scale, which is the interval $[0, 1]$, and the default value, which is 0.5. Thus, every trust relationship between a patient and a physician could be assigned by default (i.e. during bootstrapping) the value 0.5 and could take values between 0 and 1 over the system life. Also, we specify some information regarding the trustor and the trustee. In this relationship, the trustor is a human entity and has a subjective factor that influences the trust relationship: *capability belief*. This means that the belief that the patient has in the capability of the physician must be considered when assessing the trust relationship, as stated also by the *trust engine* that updates the trust relationship. This engine uses a continuous engine, meaning that it will yield a continuous value by aggregating continuous factors. The list of factors used by the engine are the reputation of the trustee, the belief of the trustor, and the trustor's quality feedback. In this quality feedback, illustrated in Figure 3.20, there is a *reputation engine* which provides target entities with reputation scores. The reputation engine gathers the claims that different patients make about a given physician and computes a final reputation using an average, which should be displayed by a *3 stars* notation. In addition to the claims, *time* is also used to derive this reputation score.

Figure 3.18: Patient-Physician relationship

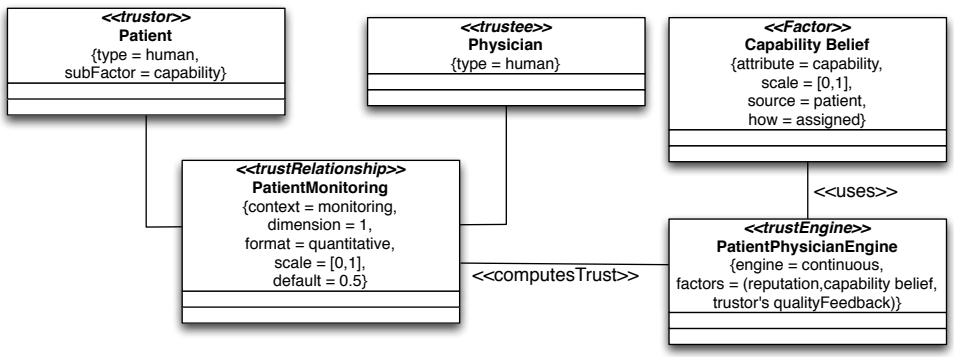


Figure 3.19: Data Retrieval Trust Relationship

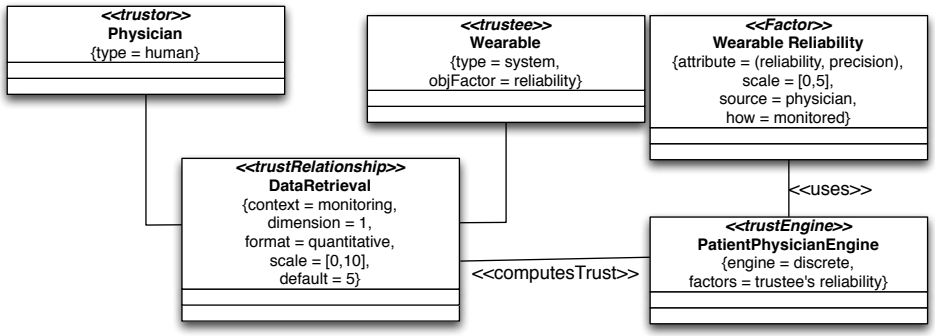
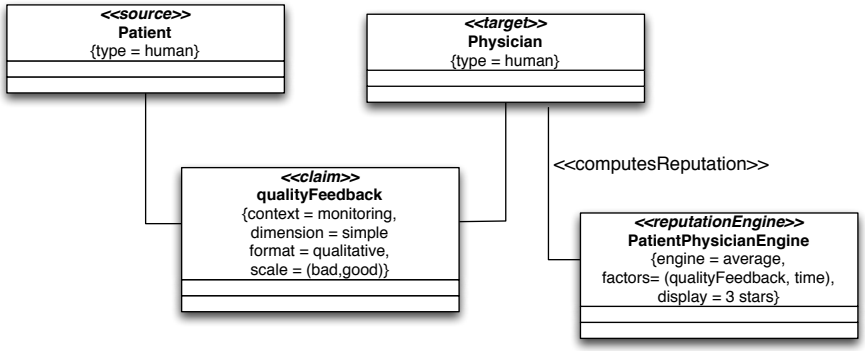


Figure 3.20: Quality Feedback Claim

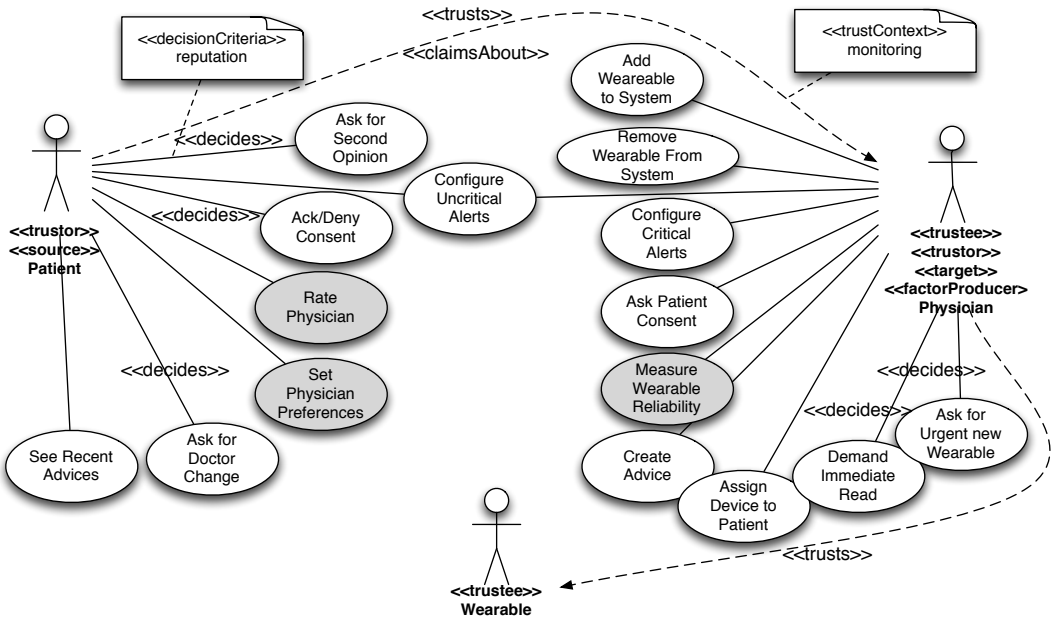


For every factor defined in the engine, we can define a new *Factor* stereotype and specify some of the important properties of them. Figure 3.18 shows that capability belief, assigned by the patient, should take a value in the interval $[0, 1]$ and should capture the attribute *trustor's capability belief*. Figure 3.19 illustrates the wearable reliability factor, which measures the reliability and precision of the wearable in a scale of $[0, 5]$, being the physician the one that triggers a system that monitors the factor.

Note that from the class diagrams information, especially after identifying the factors that we need, we can go back to the use case diagram during the second iteration and add new information as needed. The patient should have means of rating a physician and to set the physician preferences. This last use case captures the capability belief, as the preference list will likely be made by the patient in terms of this capability belief about the physicians. The physician should be able to measure the wearable reliability and update this factor, therefore he also should play the role *factor producer*. These

changes are depicted in Figure 3.21.

Figure 3.21: Trust-aware Use Case Diagram (2nd Iteration)



How the business and trust layers of the application interact may be a valuable information for designers. This can be depicted by a behavioural diagram, such as an activity diagram. The goal is to represent which actor can trigger a trust event and how, and what are the consequences of that trust event. We propose using swim lanes in order to separate the responsibilities of actors, the business logic and the trust logic in the whole application. Figure 3.22 shows the trust event triggered as a consequence of the patient changing its preference list of physicians, whereas Figure 3.23 depicts the trust event triggered when the patient asks for a second opinion.

The basic deployment for this application, without considering trust information, consists of a sensor that communicates with a wearable, which in turn, aggregates the information and sends it to a front-end server running the application. This front-end server will send the information to a back-end server that will store it into the patient's EHR and that executes a configuration application only available to administrators. Figure 3.24 shows a trust-aware deployment diagram. The wearable device can decide, based on the front-end server reputation, to which server to send information. The same happens between the front-end server and the back-end server. Of course we are

Figure 3.22: Activity Diagram for Use Case *Set Physician Preferences*

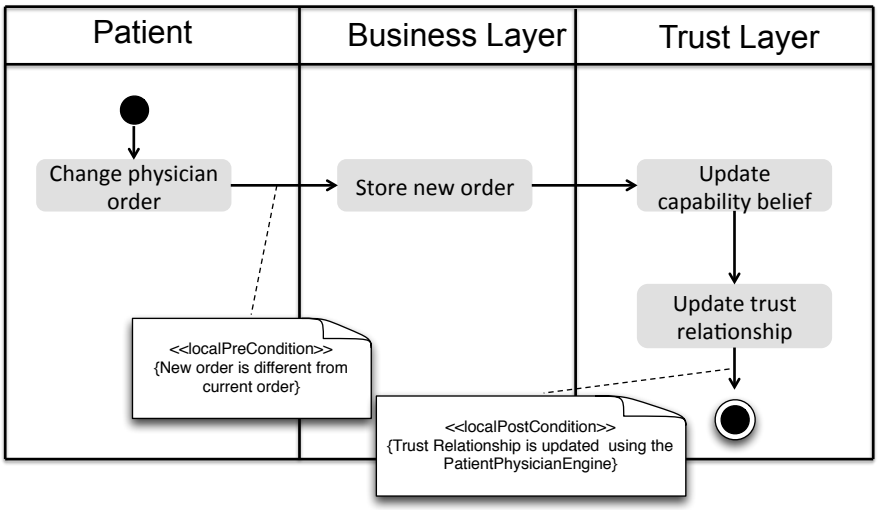
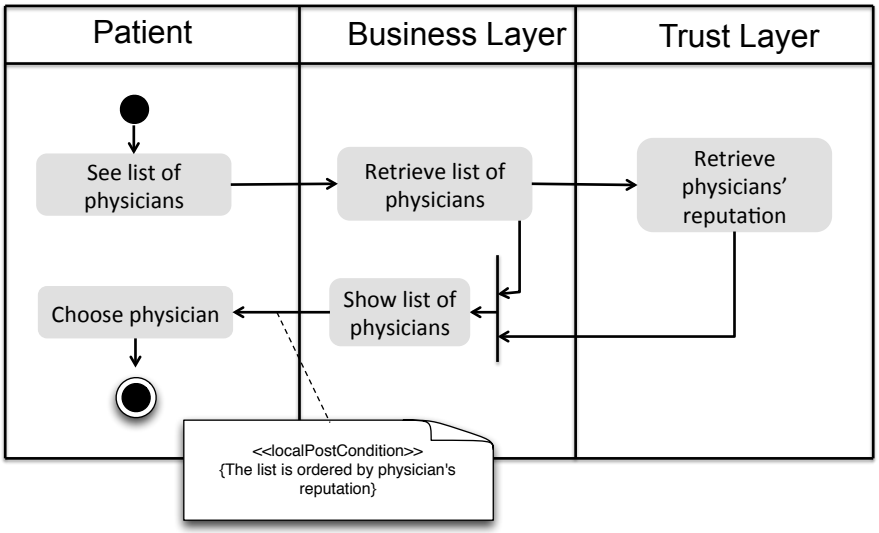
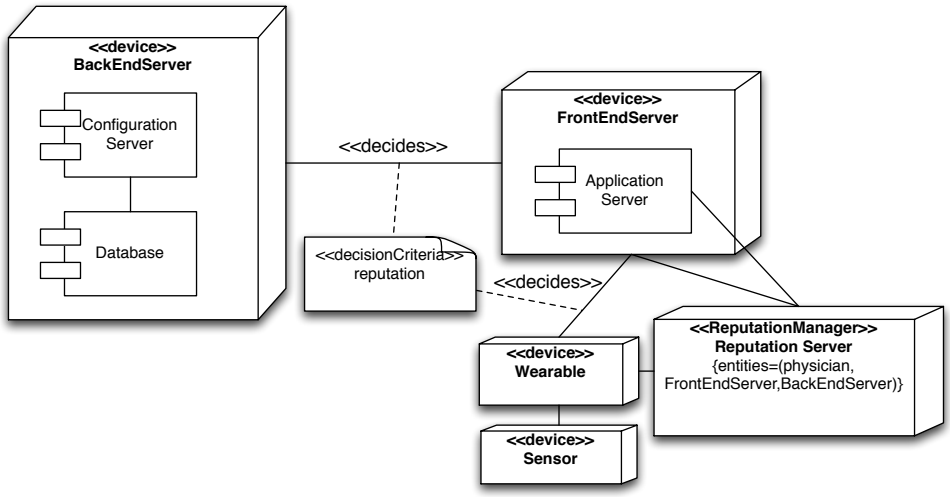


Figure 3.23: Activity Diagram for Use Case *Ask for Second Opinion*



assuming that the final deployment will consist of, at least, two front-end servers and two back-end servers. Otherwise, a decision is not possible. We can also make explicit on which node the reputation values for different entities in the system will be stored (i.e. assuming a centralized reputation model). In this case, a node is reserved to play the role of a reputation server that will store the reputation values for physicians, the front-end servers and back-end servers.

Figure 3.24: Trust-aware Deployment Diagram



3.4.5 Discussion

Our goal with this work has been to bridge the gap that prevents trust from being properly addressed during the initial phases of the SDLC. Nonetheless more work still remains to be done. First, the profile should be further extended in order to represent policies, credentials and trusted third parties, which constitute key concepts of many trust management systems nowadays, as explained in Section 2.1.2. The profile should also allow representing how trust information can be propagated between entities in the system. Trust derivation from lower software abstractions (e.g. trust among components) to higher level abstractions (e.g. trust among processing nodes), if possible at all, is an interesting field that requires much further exploration. Finally, there is a need for defining the semantics and constraints of each syntactic element. Tool support is then required to check compliance with these constraints and to derive design patterns and code from the specification. In this direction, how to integrate our approach with existing frameworks (e.g. UMLsec) should be analysed.

Chapter 4

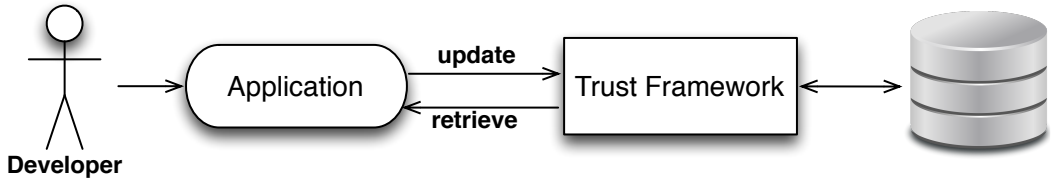
Enabling Trust and Reputation during Implementation

This chapter describes the requirements, the architecture and some implementation guidelines of a trust and reputation development framework that allows software developers to implement a wide range of trust and reputation models. The framework builds upon the concepts introduced in Chapter 2, and its goal is supporting the design and development of trust and reputation models specified by means of the tools explained in Chapter 3. The focus of this framework is on evaluation models (see Chapter 2), and therefore decision models are laid aside.

From a high-level point of view, the framework is a middle-tier server that mediates between the client application and the database tiers, as depicted in Figure 4.1. The application requests trust and reputation information from the databases, and requests updates on new trust and reputation information as a response of some events signalling. The kind of Application Programming Interface (API) that the framework exposes depends on the implementation details, and could range from remote procedure calls to a REST- or SOAP-based API. Even though we do not provide a concrete implementation, some implementation guidelines are outlined.

The chapter is organized as follows. Section 4.1 discusses the high-level requirements that the framework must meet. Section 4.2 presents a high-level architecture of the framework, which is refined into a low-level architecture in Section 4.3. Hints towards the implementation of the framework are provided in Section 4.4, and a social cloud application example is presented in Section 4.5. Finally, Section 4.6 discusses some

Figure 4.1: The Framework in Context



design and implementation aspects of the framework, as well as some challenges and lines for future research.

4.1 Framework Requirements

This section summarizes the requirements that the framework must meet. The framework has to support the implementation of evaluation models. Pure evaluation models establish trust relationships between entities, and their main goal is to compute trust values for these relationships and to help entities decide whether to collaborate or not with each other. Propagation models also build on trust relationships, and their primary goal is to disseminate existing trust information in order to derive new trust relationships. Reputation models compute reputation scores for entities, which must be stored (either centrally or distributively) and entities should be able to access this information when required.

The following list of requirements describes the coarse-grained functionality with which the framework should provide developers:

- Entities management: entities hold trust values in other entities. The framework must be able to give a unique identifier to each entity in the system and to retrieve trust and/or reputation information from an entity.

- Trust relationships management: trust relationships might change over time. New trust relationships might be created (e.g. by propagation models), other relationships might be deleted, and trust values attached to these relationships may change. Trust relationships can be affected by reputation, but also by objective and subjective factors of trustors and trustees.

- Computation engines definition: computation engines are in charge of computing a trust or reputation score, depending on the model. Although the framework can provide some default built-in metrics implementations, it is important to let developers define their own trust metrics, as they are the core concept in evaluation models.

- Events definition: events that occur in the system trigger the communication with the framework. It is required to configure the framework to respond to these events accordingly.

- Claim management: the type and value of a claim may determine a reputation score. It should be possible to configure claims in order to support application-specific needs.

- Factors management: a trust metric comprises objective and subjective factors. It is important to let developers create new factors, which can be used by user-defined metrics.

- Trust dissemination: trust information can be propagated by means of operators along trust chains. Developers should be empowered to define their own operators, or to use some built-in ones.

- Time and uncertainty: these factors may play an important role when computing a trust or reputation score. The framework should provide the developer with mechanisms to include them as part of the computation process.

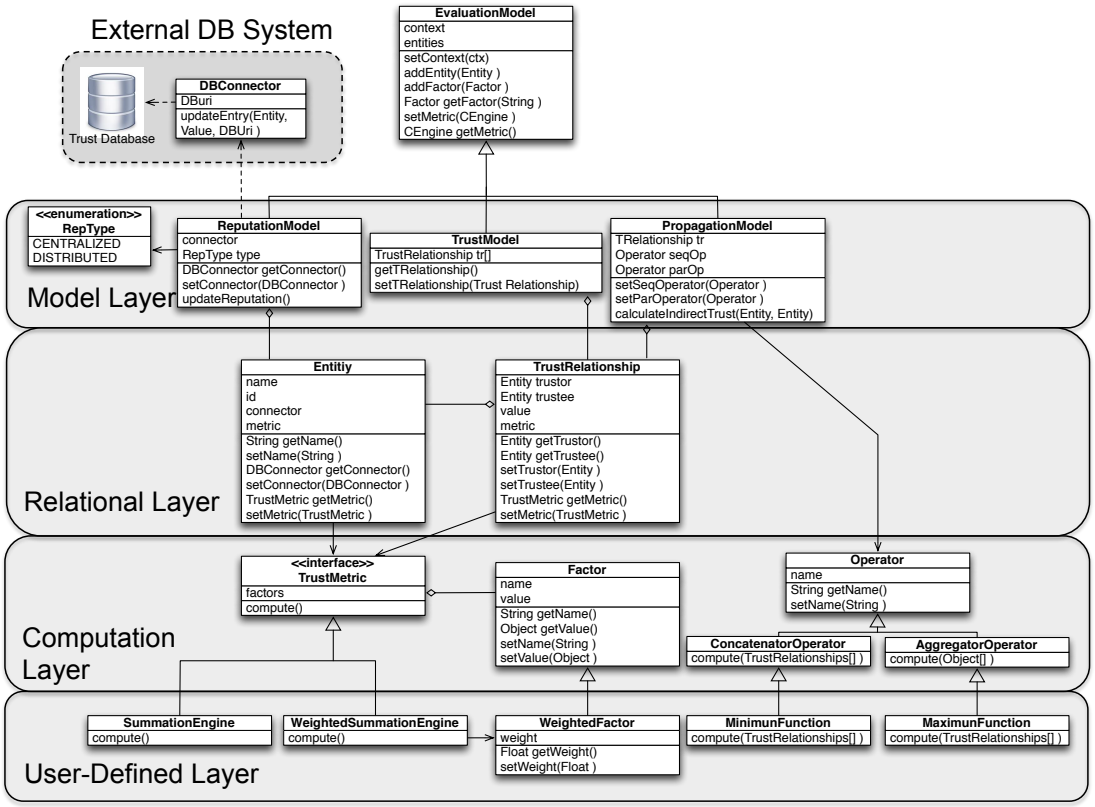
- Trust and reputation separation: the framework should allow developers to consider trust and reputation as different concepts. However, given their strong relationship, it should be possible to take each other into consideration when computing a trust/reputation score.

4.2 High-Level Architecture

A high-level architecture of the framework is depicted in Figure 4.2. This architecture bridges a gap between the conceptual model presented in Chapter 2 and the low-level architecture and implementation that will be discussed in Sections 4.3 and 4.4. Its goal

is to offer a clean separation of logic units that represent the most important concepts related to trust and reputation.

Figure 4.2: High-Level Architecture of the Framework



The architecture follows a layered design, where each layer uses the services provided by the lower layer. Likewise, the framework follows a grey-box approach, where the developer can use several functionalities in a black-box fashion as well as define new functionalities based on his needs. Next, we describe the classes and relationships for each of the layers.

4.2.1 Model Layer

In this layer we find the models that the developer can implement, namely reputation models, trust models (i.e. pure evaluation models), and propagation models. *ReputationModel*, *TrustModel* and *PropagationModel* are inherited classes from *EvaluationModels* and as such, they share a context (a string describing the context under which

the model operates) and a list of entities that take part in the model. *EvaluationModel* also provides other methods, and their functionality will be delegated to lower layer classes, depending on the model type.

A reputation model adds a connector to an external database system to store reputation scores, and it holds the type of reputation model, which might be centralized or distributed. Moreover, this class exposes the method *updateReputation*, which computes the reputation score and saves it in the trust database. A trust model contains a list of trust relationships and exposes methods to retrieve and set these relationships. Finally, a propagation model, in addition to containing a list of trust relationships, it also contains concatenator and aggregator operators, and exposes a method to calculate indirect relationships.

4.2.2 Relational Layer

This layer contains the basic building blocks on which the models of the upper layer rely: entities and trust relationships.

Entities have a name, an automatically-generated identifier, a database connector and a trust metric. The fact that each entity holds a database connector enables distributed reputation systems, where each entity must store the reputation information regarding another entity in a personal database. Likewise, as each entity holds a trust metric instance, we allow each entity in the model to use a different trust metric to compute other entities' reputation.

Regarding trust relationships, they consist of a tuple that specify which is the entity that places trust (trustor), the entity on which trust is placed (trustee), the extent to which the trustor trusts the trustee (value), and the trust metric used to derive this value.

The decision that both an entity and a trust relationship may define their metrics supports the implementation of more advanced trust models where the final trust value that a trustor places on a given trustee might be determined by both, the reputation of the trustee, and the trust relationship between the trustor and the trustee.

4.2.3 Computation Layer

Evaluation models rely on trust metrics to perform trust values calculations. This is the layer in charge of such computation.

TrustMetric is an interface that a developer should implement to override the *compute()* method, where the trust calculation takes place. Trust metrics use factors, through the class *Factor*, which have a name and a value, as well as methods to retrieve and set these parameters. Operators for propagation models belong also to this layer.

Note that trust metrics contain instances of factors. As entities and trust relationships hold in turn instances of trust metrics, each entity or relationship might use different factors, increasing the flexibility of the framework to accommodate complex models.

4.2.4 User-Defined Layer

This layer is created when users extend the computation layer to accommodate their own definitions. Users can create new computation engines (implementations of the *TrustMetric* interface) and new factors to implement a wide range of models. For illustration purposes, the architecture includes a summation engine (that basically sums up the factors that it contains) and a weighted summation engine (that adds a weight to each factor). The latter requires creating a specialized factor class that adds the weight to its internal state.

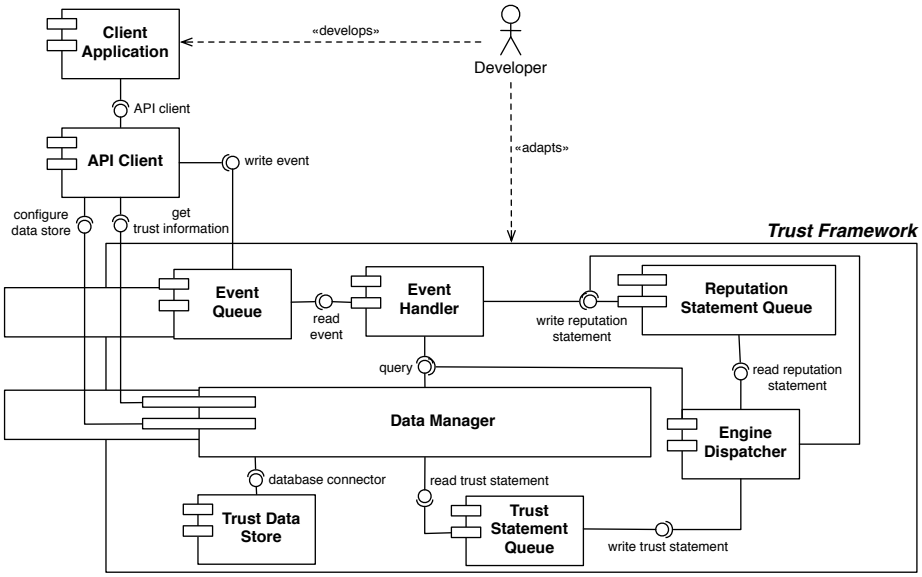
4.3 Low-Level Architecture

In the previous section, we have described the main elements and their relationships from a logical and layered point of view. This section provides deeper insight and includes elements that are close to the implementation of pure evaluation and reputation models. Some considerations for the implementation of propagation models are discussed in Section 4.6.

The low-level architecture is shown in Figure 4.3. The framework API provides three interfaces: *write event*, to explicitly signal that an event has occurred, *get trust information*, to retrieve trust-related information, such as the reputation of a given entity or existing trust relationships. Additionally, the API may provide mechanisms to configure the trust database attributes and tables.

There are three components that represent queues: the *Event Queue*, which stores events, the *Reputation Statement Queue*, which stores reputation statements and the *Trust Statement Queue*, which stores trust statements.

Figure 4.3: Framework Architecture: Component Diagram

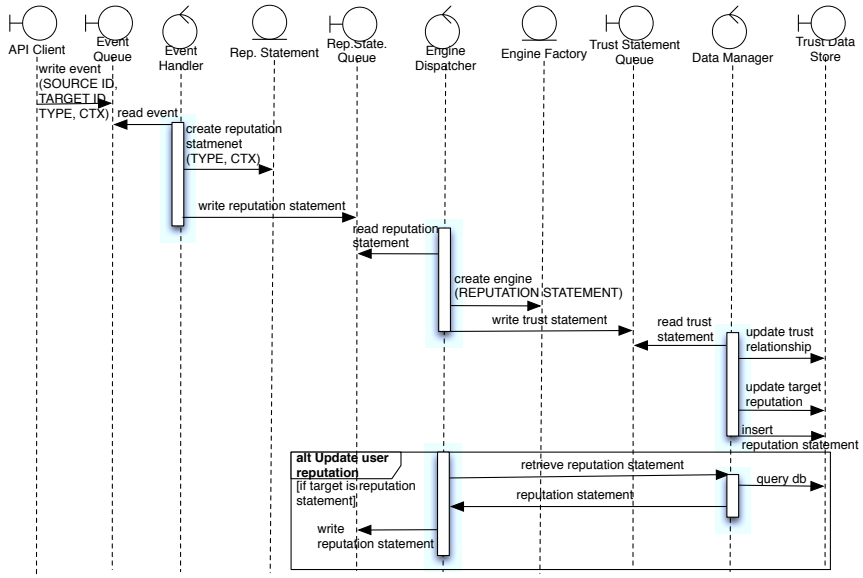


The *Event Handler* component is in charge of reading events from the *Event Queue*, creating reputation statements according to the event type and writing these events into the *Reputation Statement Queue*. The *Engine Dispatcher* component reads reputation statements and creates engines that perform computation on these statements, producing trust statements. The *Data Manager* component prevents the rest of components from needing to know the internal details of the database schema and technology used. It basically transforms queries and write actions into technology-dependent statements (e.g. Structured Query Language (SQL) statements). The *Trust Data Store* component represents the Relational Database Management System (RDBMS) that provides persistence to the framework.

Figure 4.4 depicts a coarse-grained sequence diagram that describes the steps triggered by an event signalled by the client application. Note that there are two architectural elements that are not shown in the component diagram, but have been added here for a clearer understanding. The *Reputation Statement* entity represents a reputation statement, that is, a tuple $< source, claim, target >$. The other entity is *Engine Factory*, which is used by the *Engine Dispatcher* in order to create an appropriate engine according to the reputation statement type.

The bottom part of Figure 4.4 describes the steps carried out if the target of a

Figure 4.4: Framework Architecture: Sequence Diagram



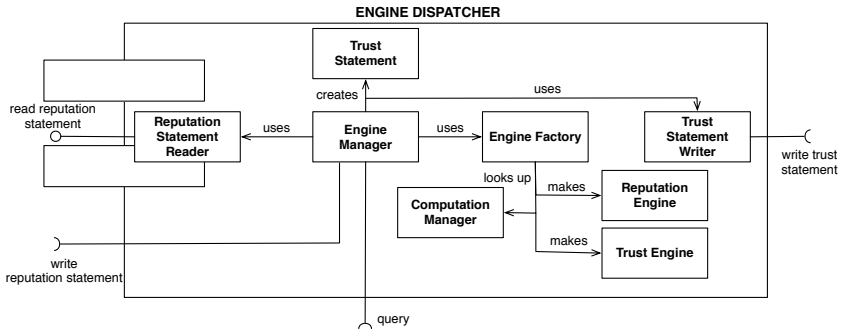
reputation statement is, in turn, a reputation statement. In this case, the original reputation statement is retrieved from the database through the data manager component and a new reputation statement will be written in the *Reputation Statement Queue* for later processing. In a real scenario, this typically happens when a user states that the information provided by another user (that is, a reputation statement), has been helpful. In this case, the reputation of the reputation statement would be updated, but the reputation of the user who provided the information could be updated as well. This behaviour could be configured by the developer, for instance, by an optional parameter passed as an argument.

4.3.1 Components Decomposition

The *Engine Dispatcher*, *Event Handler* and *Data Manager* components are decomposed into modules in Figure 4.5, Figure 4.6 and Figure 4.7 respectively.

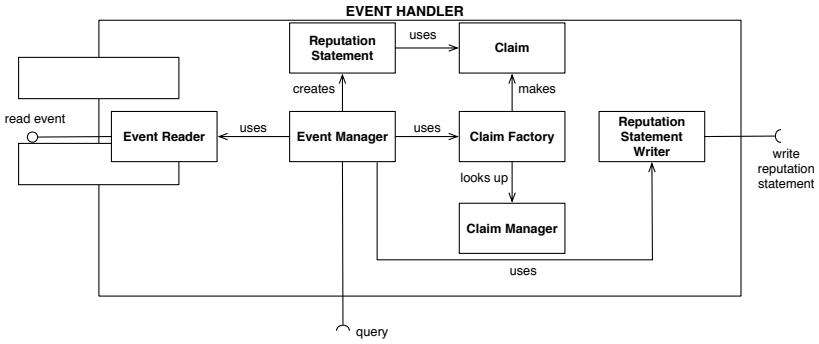
In the case of the *Engine Dispatcher*, the *Engine Manager* module uses the *Reputation Statement Reader* module in order to retrieve a reputation statement. It also uses an *Engine Factory*, which accesses a *Computation Manager* in order to figure out how to make two engines: a *Reputation Engine* and a *Trust Engine*. The manager creates a *Trust Statement* and uses a *Trust Statement Writer* that knows to which queue to for-

Figure 4.5: Engine Dispatcher Component



ward a trust statement. Finally, the manager allows querying information and writing reputation statements into queues in order to cover the case in which the target of the reputation statement is itself a reputation statement.

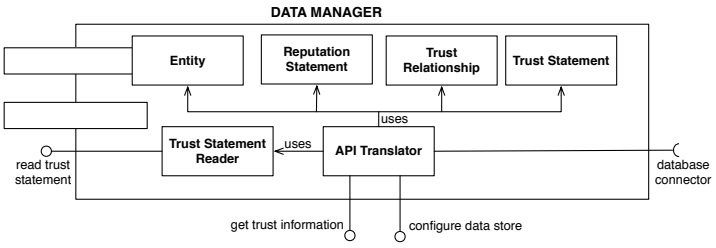
Figure 4.6: Event Handler Component



The *Event Handler* uses an *Event Reader* in order to retrieve events from an event queue, and a *Claim Factory* in order to create a *Claim*. The factory inspects a *Claim Manager* to determine which type of claim to make. The manager creates a *Reputation Statement*, which uses a *Claim* and a *Reputation Statement Writer* in order to forward a reputation statement to a queue.

As for the *Data Manager*, it has an *API Translator* module that uses the *Trust Statement Reader* in order to retrieve a trust statement and translates the native API call (e.g. Java) into a technology-dependent statement (e.g. SQL) through a database connector. The API translator may encapsulate information in data structures (e.g trust relationship) that will be sent back to other components or the client application.

Figure 4.7: Data Manager Component



4.3.2 Data Structures

Even though more modules and structures might arise in a further detailed design, there are four data structures that are specially relevant as they encapsulate crucial information that flow between components. If we consider them from an Object-Oriented (OO) perspective, they could be refined as classes, which are shown in Figure 4.8. For each of these structures, only the most important attributes and applicable functions (i.e. methods in OO design) are shown.

The *Event* structure represents an event by means of a name, a context, a source of the event, and a target of the event. Several event types can be pre-defined, but new events can be created.

A *Claim* represents the assessment made by an entity. The type of event that is triggered determines the type of claim that is made. A claim has a scale (minimum and/or maximum boundaries) and a value, which might be numeric or qualitative. A claim can be normalized (resp. denormalized) from its range scale (resp. interval $[0, 1]$) to the interval $[0, 1]$ (resp. its range scale).

A trust metric comprises a set of subjective and objective *Factor*s. Two typical subjective factors are introduced in the next section, but there might be more factors that the trust metric may require. A factor is identified by a name, a value, a source entity and a target entity. In some situations a factor may represent a property or aspect of an entity that is independent of any other entity (e.g. an objective factor about an entity, such as the number of transactions that the entity has completed), and therefore the factor will only have a target entity.

A *Reputation Statement*, as stated previously, contains a source, a claim and a target. Source and target are entities. In order to allow developers to take time into

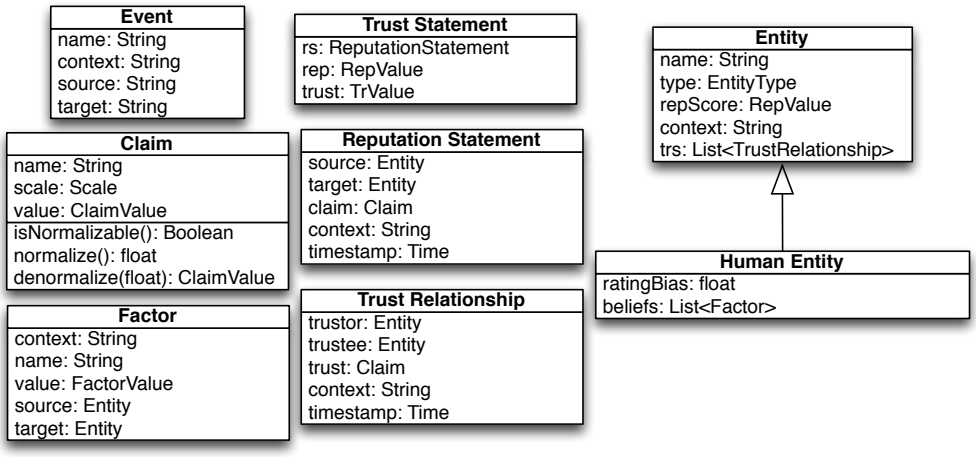
account, a reputation statement holds a time stamp, which indicates when the reputation statement was made. Also, as a reputation statement is made in a context, the latter is considered as part of the statement.

A *Trust Statement* (which is a new notion that has not been mentioned previously) contains a reputation statement, a reputation value and a trust value. This structure allows the framework to convey trust and reputation information separately, fostering the idea that trust and reputation are two different concepts. A trust statement is the structure that is passed onto the data manager in order to update the different database tables.

A *Trust Relationship* represents the trust value that a trustor (the source *Entity*) places on a trustee (the target *Entity*). As in the case of reputation statements, trust relationships need to consider time and the context under which they make sense.

An *Entity* represents any object that can be evaluated. It has a unique name, a type (Human, Non-Human and Reputation Statement), a reputation score and the context under which the reputation score is assigned. A *HumanEntity* is an entity which, additionally to the previous fields, also holds a rating bias and a list of beliefs, which can be actually represented by factors.

Figure 4.8: Important Data Structures



4.3.3 Incorporating Trustor's Subjective Factors

When an entity rates another entity, their trust relationship may change. As explained in Chapter 2, there are other factors that influence trust beyond reputation, such as the trustor's subjective properties. The framework provides developers with support to include the following properties:

- Rating bias: this property indicates the usual disposition of an entity to high or low ratings. If the entity has always rated others with the maximum value in the past, the fact that now the same entity rates a new entity with a high value does not give much information. However, it would give a lot of information if the entity rated another entity with a low value. There are different ways to provide built-in support for this. One way is by using standard deviation of all the claims made by the entity over a period of time.

- Beliefs: they indicate how much an entity believes in the capability, honesty, etc. of a target entity. This information could be directly assigned by the entity, or it could be derived automatically from several events depending on the context. In a social network application, for example, the number of visits to the target's profile or the ratings given to other entities' claims could provide insightful information for determining beliefs among entities. Beliefs can be represented as factors data structures (see previous section) where the source entity holds the belief about the target entity.

The next section provides further details on possible implementation options.

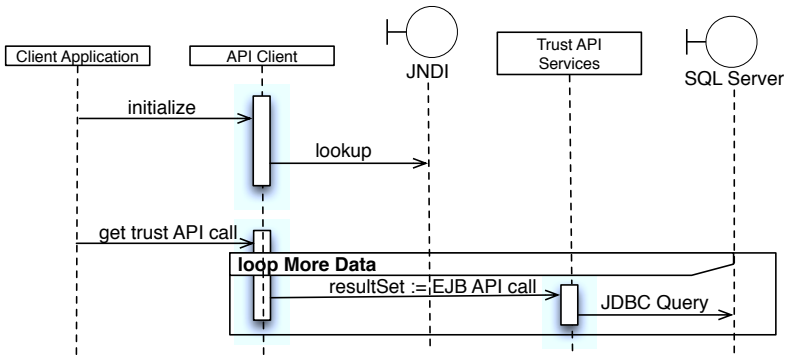
4.4 Implementation Guidelines

The framework can be deployed as a JavaEE[1] application that constitutes a runtime platform onto which to develop trust-aware applications. The hint of implementing the framework in Java is two-fold: since Java is a quite popular development language, it may be easier for developers to familiarize with the framework and with the mechanisms to adapt it to their needs. Furthermore, we achieve portability, as the framework can be executed on any platform and operating system.

[1] `http://docs.oracle.com/javaee/7/index.html`

As an example, Figure 4.9 shows the steps that a client application would perform in order to query trust-related information from the trust database. The client application would need to look up an instance of the trust server from the Java Naming and Directory Interface (JNDI) in order to invoke the query API call, implemented by EJBs (Enterprise Java Beans) that connect to an SQL server by means of a Java Database Connectivity (JDBC) connector.

Figure 4.9: Query API Call Sequence Diagram



In the next sections, we elaborate on some implementation ideas for different architectural elements and concepts.

4.4.1 Context

As we mentioned in Section 2.2.3, the context is very important in the trust and reputation domains. Every trust relationship and reputation score make sense in a single context and cannot (usually) be transferred directly to another context. In order to take into account the context, whenever an event is triggered, the developer introduces a string that represents this context. It will then be stored together with all the rest of trust or reputation information, as explained in the next section.

4.4.2 Database Tables

An RDBMS such as SQL can be the implementation choice in order to store persistently all the trust-related information. The tables design is of paramount importance for the efficiency and correct behaviour of the framework, as they may support more or less easily the implementation of the concepts discussed above. As an example, we propose

Table 4.1, Table 4.2, Table 4.3 and Table 4.4. For each one, we explain the main attributes they should hold and their meaning.

Table 4.1: Entity Table

Attribute	Description
ID	Unique identifier
Type	Human, Non-Human or Reputation Statement
Reputation	Reputation score
Context	Context where this information applies
Claim	Type of the claim that corresponds to this evaluation
Time	Indicates when this reputation score was first created
LastTime	Indicates the last time this entity's reputation was changed
Number of evaluations	Number of evaluations made on this entity

Table 4.2: Trust Relationship Table

Attribute	Description
ID	Unique trust relationship identifier
ID Trustor	The unique identifier of the entity placing trust
ID Trustee	Unique identifier of the entity onto which trust is placed
Trust value	Trust value placed by the trustor on the trustee
Context	Context where this trust relationship holds
Number of updates	Times that the value of this relationship has changed
Time	Indicates when this trust relationship was first established
Last Time	Indicates when this trust relationship was last updated

Primary keys could be a made up of the ID and the Context, since the same entity could hold different trust or reputation values for different contexts. Unique identities could be the foreign keys used in order to relate tables among each other.

The *Data Manager* component must provide the interface required to set and obtain most of this information. For this purpose, it uses a JDBC connector in order to translate from Java method calls to SELECT, INSERT and UPDATE SQL statements. Given that this can be complex, a suitable, more maintainable design approach would

Table 4.3: Reputation Statement Table

Attribute	Description
ID	Unique identifier
Source	Source entity's ID
Target	Target entity's ID
Claim	Name of the claim
Claim value	The value of the claim
Context	Context where the statement is applicable
Time	Time when the claim was made

Table 4.4: Beliefs Table

Attribute	Description
ID	The unique identifier of the belief (e.g. capability, honesty, ...)
Source	Source entity's unique ID
Target	Target entity's unique ID
Value	Belief value
Context	Context where this belief is applicable

be to create different objects to manage each table. The *API translator* would be split into a mediator object that delegates the queries to these specialized objects. Thus, one object would not need to know how to interact with all the database tables.

Note that these tables support taking the trustor's subjective factors into account. In order to compute the *rating bias* of an entity, the *Data Manager* would first retrieve all the claims made by the entity, normalize them (in order to consider different types of claims with different scales), and then compute the average and the standard deviation. This can be achieved by looking up the *Reputation Statement* table (see Table 4.3).

Another implementation choice would be using an Object-Oriented Database Management System (OODBMS), which provides higher flexibility and avoids the tedious mapping between two representation models (i.e. from objects to relational tables), as they allow storing objects directly. This simplifies greatly the implementation of the *Data Manager*, which would basically become a direct mediator between an API call

and the database system, without requiring the translation process. However, RDBMS are often more efficient, above all when considering simple objects and relationships.

4.4.3 Messaging Infrastructure

The main components of the architecture communicate with each other via asynchronous, optimistic queue-based messaging system, which could be implemented onto the Java Message Service (JMS). This solution scales well in the presence of many entities, which can continue their operations in most cases without the need to wait for results from the framework. Whenever a new event is triggered, the client application does not wait for a response from the server, but it continues doing other tasks. The application knows that the trust server will eventually update the trust relationships and reputation scores, but there is no hard time limit.

The same happens in the case of the *Event Handler*, *Engine Dispatcher* and *Data Manager* components. They are listening to their specific queues. As soon as a new piece of information arrives, they take it out from the queue, process it, and place it in another queue for further processing. This way, the framework can adjust, on-demand, the number of instances of the same component that is listening to a queue, providing higher performance and scalability. This is especially true for the *Data Manager*, which can receive queries and writes requests from different components: the *Trust Statement Queue*, the *Engine Dispatcher*, the *Event Handler*, and even from the client application. Therefore, many instances could be concurrently listening to queries from these sources.

4.4.4 Engines

When an *Engine Dispatcher* instance reads a reputation statement, it uses an *Engine Factory* to create the appropriate *Engine* to deal with such statement. The *Engine Factory* creates an engine by inspecting computation rules, which define which type of engine to create under which circumstances. This can be implemented as an XML file that the factory reads. This file includes a set of conditions and an effect, which states the engine type to build. A simple example is shown in Listing 4.1:

Listing 4.1: Engine Configuration

```
1 <CompRule RuleId=``CompRule CounterUp'' Engine=``CounterUp''>
2     <Context>Film Review</Context>
3     <Claim>Positive Vote</Claim>
```

```
4     <Source>
5         <SourceType>Human</SourceType>
6     </Source>
7     <Target>
8         <TargetType>Non-Human</TargetType>
9     </Target>
10 </CompRule>
```

This file specifies that if the context of the reputation statement is *Film Review*, the claim is *Positive Vote*, the type of the source of the reputation statement is *Human*, and the type of the target of the reputation statement is *Non-Human*, then the engine to apply is *CounterUp*. The *Engine Factory* would read the file (through another XML reader object), compare the conditions against the reputation statement fields, and create an instance of this type of engine.

Listing 4.2 shows how the class *EngineManager* would work.

Listing 4.2: Excerpt of EngineManager

```
1  public final class EngineManager {
2
3      //...more stuff
4
5      //A Reputation Statement Reader instance signalled that a new
           reputation statement has arrived.
6      public void onNewReputationStatement(ReputationStatement rs) {
7          Engine[] e = EngineFactory.getEngine(rs);
8          //e[0] holds an instance of the reputation engine
9          //e[1] holds an instance of the trust engine
10         if (e[0] != null) {
11             reputation = e[0].compute(rs);
12         }
13         if (e[1] != null) {
14             trust = e[1].compute(ts);
15         }
16         //tsw is an instance of a trust statement writer, a JMS client
17         //which actually knows how to send data to which queue
18         tsw.write(new TrustStatement(rs,e[0],e[1]));
19     }
20 }
```

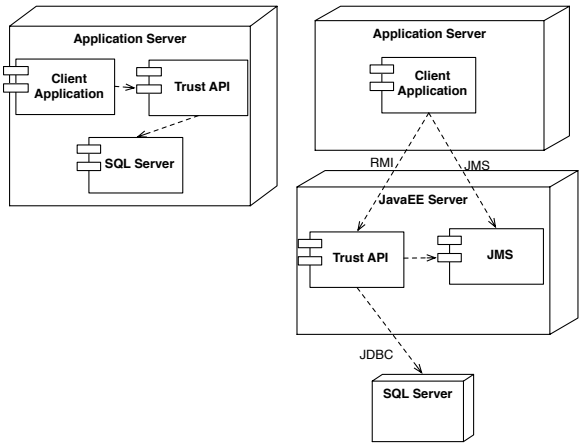
4.4.5 Deployment

The decision on how deployment is done is of paramount importance when pursuing high performance behaviour in environments with thousands of entities. There are several choices:

- Everything on the same machine: this is the simplest deployment option. It does not scale and does not allow for failover capabilities in case of electrical problems.
- The application server on one machine, the trust server and the RDBMS server on other machine: an intermediate solution where the trust server can be replicated on-demand, offering a higher scalability.
- Everything on different machines: this is the most flexible choice, although it is more vulnerable to network and bandwidths problems.

As an example, Figure 4.10 shows the first and the third deployment options discussed above.

Figure 4.10: Two Deployment Configurations



The next section ties together all the concepts discussed here by showing how the framework can be applied in a social cloud scenario.

4.5 Application Example: Social Cloud

This section presents a motivating scenario that would benefit from the use of the presented framework. We describe the scenario, its trust and reputation requirements

and how the framework can be used in order to implement these requirements.

4.5.1 Scenario Description

The scenario that will be used as benchmark to validate our framework is the following. A developer needs to implement a social website for cloud providers. Cloud providers can register in the site. Once registered, they can publish web services on the site by posting a full description of the service, including the API calls (e.g. by using Web Service Description Language (WSDL)). Cloud providers can also look up a web service according to their needs, and use the service in order to create a larger, composed web service. When one cloud provider consumes a service from another provider, the latter can charge the former according to the type or complexity of the service. Thus, the site acts as a software market between cloud providers, following the *software as a service* model. Eventually, each cloud provider will use its own infrastructure to provide the resulting services to their customers, although this is out of the scope of the scenario.

Figure 4.11 depicts the main elements of the scenario together with trust and reputation considerations that are further explained in the following section.

4.5.2 Trust Requirements

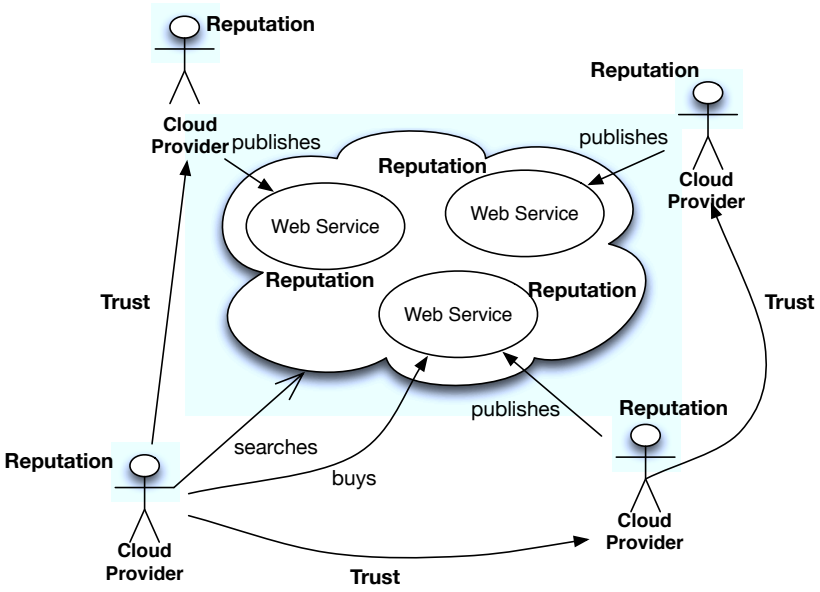
The underlying framework must enforce trust and reputation requirements in order to prevent risks for the cloud providers and to foster trust in the site.

There are two basic reputable entities in this scenario: cloud providers and web services. Each of them might have a reputation value that can be derived from the personal opinions and feedbacks from other providers in the site. For example, if a provider uses a service and notices that the service is not running appropriately, it could rate negatively the service, which in turn could negatively affect the service provider's reputation.

In addition to reputation, cloud providers can establish trust relationships among themselves. The way trust and reputation are computed depends on the models implemented, and the developer of the website should be provided with mechanisms to decide which model to use at design-time.

For a particular instance of this scenario, we focus on a small subset of possible trust requirements:

Figure 4.11: Social Cloud Scenario



1. Cloud providers can rate web services with one, two or three stars. When cloud providers rate a web service, this affects the reputation of the web service and the trust relationship between the evaluator and the web service creator.

2. Cloud providers can rate each other by using a *I like* and *I don't like* statement. When cloud providers rate other providers, this affects only the trust relationship between them, but not the reputation.

3. Reading a cloud provider profile increases the capability belief of the reader.

4. Reputation scores in the context *WebServiceForOffice* must be normalized to the range $[0, 1]$ prior to being written in the trust database, and must be denormalized to the original range of the model prior to being sent to the application.

The next section discusses how we can implement these requirements by using the framework.

4.5.3 Implementation

The realization of the first requirement would be as follows. As the framework updates trust information after an event has occurred, the first step for the developer is deciding

the name of the event. Then, he must decide the claim type that is associated to this event. In this case, the chosen event name is *WebServiceRating*. Listing 4.4 shows how a new claim type could be created.

Listing 4.3: Creating a Bounded Claim

```
1  //BoundedClaim is a claim that is (de)normalizable according to
2  //a linear transformation. This claim would be offered by default.
3  public class BoundedClaim extends Claim {
4
5      int value, minimum, maximum;
6      String name;
7
8      BoundedClaim(String n, int v, int min, int max) {
9          name = n;
10         value = v;
11         minimum = min;
12         maximum = max;
13         isNormalizable = true;
14     }
15
16     public float normalize() {
17         //Applies a linear transformation
18         //[minimum,maximum] -> [0,1]
19     }
20     public int denormalize() {
21         //Applies a linear transformation
22         //[0,1] -> [minimum,maximum]
23     }
24 }
```

Then, it is required to bind the claim to the event. This can be done by configuring an XML file as shown in Listing 4.4.

Listing 4.4: Binding Claim Type and Event

```
1  <EventClaim evId=''EV Example'' Engine=''CounterUp''>
2      <Event>
3          <Name>WebServiceRating </Name>
4      </Event>
5      <Claim>
6          <Name>WebServiceRating</Name>
7          <Class>BoundedClaim</Class>
```

```
8         <Min>1<Min>
9         <Max>3<Max>
10        <Value>EVENTVALUE</Value>
11    </Claim>
12  </EventClaim>
```

The developer is specifying that when an event with name *WebServiceRating* is triggered, a *BoundedClaim* object must be created with the parameters name, minimum, maximum and value. The value parameter is given by the value parameter in the event object.

The next step is creating the trust and reputation engines. This is done by extending the *Engine* abstract class and implementing the *compute()* method, as shown in Listing 4.5. Note the use of Java generics[2] to specify the type of the values returned by the engines.

Listing 4.5: Implementing Trust and Reputation Engines

```
1  public class RepExampleEngine extends Engine<Float> {
2
3     //It computes the target's reputation by multiplying the
4     //claim value by the reputation of the source
5     public Float compute(ReputationStatement rs) {
6        return rs.getClaim().getValue() * rs.getSource().getReputation
             ();
7     }
8  }
9
10 public class TrustExampleEngine extends Engine<String> {
11
12    //It computes the trust value between the trustor and the trustee
          by multiplying
13    //the trustor's belief in the target's capability by the claim
          value.
14    public String compute(ReputationStatement rs) {
15       List<Factor> lb = rs.getSource().getBeliefs();
16       float capBelief = retrieveFactor(lb, "capability", target);
17       float aux = capBelief * rs.getClaim().getValue();
18
19       if (aux > THRESHOLD) {
```

[2] http://docs.oracle.com/javase/tutorial/java/generics/types.html

```
20              return 'TRUSTWORTHY';
21          } else {
22              return 'UNTRUSTWORTHY';
23          }
24      }
25
26      private float retrieve(List<beliefs> lb, String name, HumanEntity
            target) {
27          //This method retrieves the value of the belief with name '
                name'
28          //about an entity 'target' from the list of beliefs 'lb'
29      }
30  }
```

Finally, the developer should configure the computation rules by XML, as shown in Listing 4.7.

Listing 4.6: Configuring Engines

```
1  <CompRule RuleId=``CompRule Example'' RepEngine=``RepExampleEngine''
       TrustEngine=``TrustExampleEngine''>
2      <Context>
3          <Name>any<Name>
4      </Context>
5      <Claim>
6          <Name>WebServiceRating</Name>
7      </Claim>
8  </CompRule>
```

The developer is specifying that, no matter which the context is, if the name of a claim is *WebServiceRating*, then apply *RepExampleEngine* and *TrustExampleEngine* as reputation and trust engines respectively. Once all this is configured, the developer only needs to retrieve a trust server instance through JNDI and to make the corresponding API call, where *eventName* should be *WebServiceRating*:

Listing 4.7: API Call for Sending the Event

```
1  public void sendEvent(String eventName, String ctx, String source,
       String target)
```

A detailed sequence diagram of this use case is depicted in Figure 4.12 and Figure 4.13. The server, upon receiving the call, creates the *Event* and sends it to the

Figure 4.12: Detailed Sequence Diagram of *sendEvent* API Call

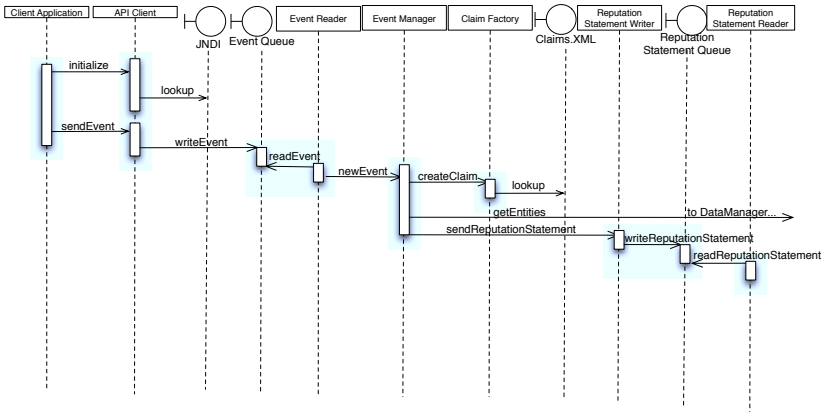
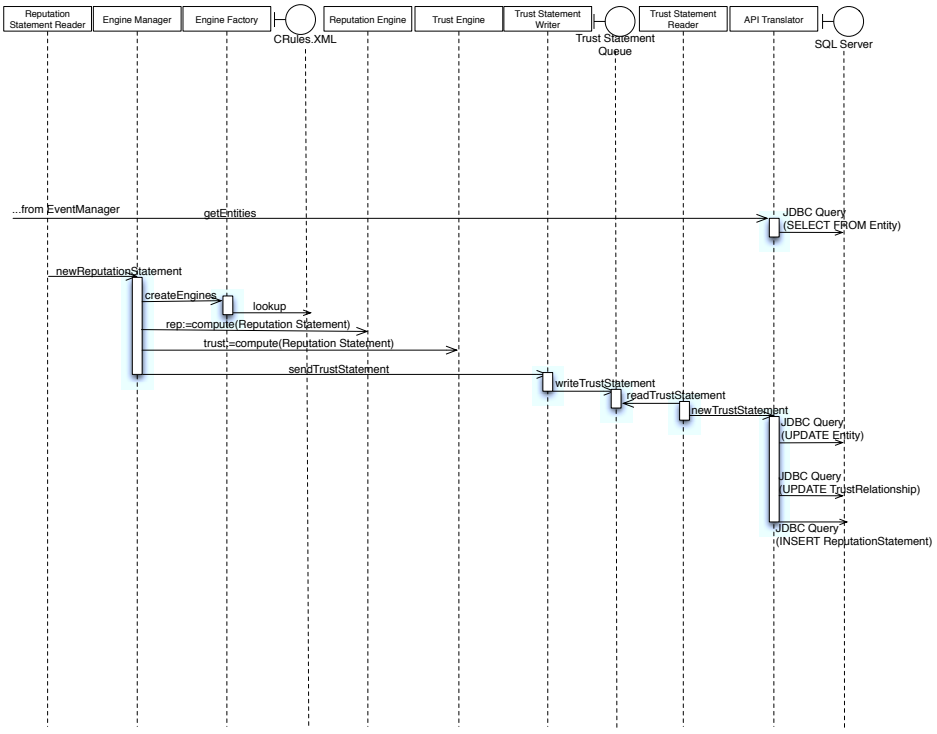


Figure 4.13: Detailed Sequence Diagram of *sendEvent* API Call (cont.)



Events Queue. From that moment, the client application can continue its execution. An *Event Manager* instance is asynchronously notified by an *Event Reader* that a new *Event* is available for processing, and it creates a *Claim* according to the rules specified in the XML file. The *Event Manager* also retrieves the complete information about

the entities using a *Data Manager* instance, and with all this information it creates a *Reputation Statement*, which is sent to the *Reputation Statement Writer*, which in turn writes it in the *Reputation Statement Queue*.

The *Engine Manager* is asynchronously notified by a *Reputation Statement Reader* instance that a new *Reputation Statement* is in the queue ready for processing, and it creates an *Engine* using the computation rules codified in the XML file. The values returned by the reputation and trust engines are encapsulated in a *Trust Statement* instance by the *Engine Manager*, and sent to a *Trust Statement Writer* instance, which writes it into the *Trust Statement Queue*.

Finally, the *Data Manager* is notified by a *Trust Statement Reader* instance upon the arrival of the new *Trust Statement*. Using the JDBC connector, it translates the *Trust Statement* into the appropriate INSERT/UPDATE statements in the *Entity*, *Reputation Statement* and *Trust Relationship* tables.

An instantiation of this use case could be as follows. Let us assume that a cloud provider *CP1* rates the web service created by another provider *CP2* with 3 stars. The resulting tables after this event occurs are shown in Table 4.5, Table 4.6 and Table 4.7.

Table 4.5: Entity Table Example

Attribute	CP1	WebService	RepStatement
ID	CP1-ID	CP2-WebService0123	RepStatementCP1-CP2
Type	Human	Non-Human	RepStatement
Reputation	0,7	2,1	-
Context	WebServiceRatingCtx	WebServiceRatingCtx	WebServiceRatingCtx
Claim	AnotherClaim	BoundedClaim	-
Time	10-05-Mon25Sep2012	14-30-Tue05Dec2011	-
LastTime	10-05-Mon25Sep2012	09-24-Wed24Jan2012	-
Number of evaluations	1	5	0

The second requirement is similar to the previous one, therefore the steps are also similar. However, there are two major differences. The first one is that the value of the new claim is set to *null* (there is no value in a *I Like* or *I don't Like* statement) and therefore it is not possible to normalize it. The second one is that the reputation engine is set to *null* in the computation rules XML file, since no reputation update is required.

Table 4.6: Trust Relationship Table Example

Attribute	CP1 - CP2
ID	CP1-CP2-RelID
ID Trustor	CP1-ID
ID Trustee	CP2-ID
Trust value	TRUSTWORTHY
Context	WebServiceRatingCtx
Number of evaluations	1
Time	09-24-Wed24Jan2012
Last Time	09-24-Wed24Jan2012

Table 4.7: Reputation Statement Table Example

Attribute	RepStatement
ID	RepStatementCP1-CP2
Source	CP1
Target	CP2-WebService0123
Claim	BoundedClaim
Claim value	3
Context	WebServiceRating
Time	09-24-Wed24Jan2012

The developer, upon detecting that a cloud provider has rated another provider, would call the *sendEvent* call with the new event name provided by himself.

Regarding the third requirement, updating trust-related information (such as beliefs, or other types of factors) does not require the developer to trigger an event, but only to use the API call that requests the *Data Manager* to update this information. Therefore, upon detecting that a cloud provider has seen other provider's profile, it would perform a call similar to the following:

Listing 4.8: Changing a Belief

```
1 public void changeBelief(String beliefName, String ctx, String
    source, String target, float increaseValue)
```

As for the last requirement, the framework should offer hot spots or hooks in order to provide developers with extension points for application-specific needs. The methods of an abstract class *InfoFilter* can play this role, as depicted in Listing 4.9.

Listing 4.9: Framework Hooks

```
1  abstract class InfoFilter {
2
3      //This method is called right before an engine receives a
           reputation statement
4      public ReputationStatement beforeComputation(ReputationStatement
           rs);
5
6      //This method is called right after an engine computes a trust
           statement
7      public TrustStatement afterComputation(TrustStatement ts);
8
9      //This method is called right after retrieving some trust
           information
10     //and right before sending this information to the client
           application
11     public RepValue afterRetrieval(RepValue rv);
```

Developers need to extend this class and implement the methods according to their needs. In the case of the last requirement, a reputation score in the context *WebServiceForOffice* must be normalized in the range $[0, 1]$, and denormalized to the original range prior to being sent back to the application. Therefore, the developer would need to place the normalization code in the *afterComputation()* method, and denormalization code in the *afterRetrieval()* method.

4.6 Discussion

We have presented a trust framework that assists developers to implement applications that need to take into account trust and reputation requirements. This kind of applications are steadily emerging, responding to an increasing demand of users eager to participate in collaborative environments. We have seen this trend with the success of blogs and social networks.

4. ENABLING TRUST AND REPUTATION DURING IMPLEMENTATION

In spite of their importance, trust and reputation are often discussed in the literature from a theoretical perspective: hundreds of trust and reputation models can be found, but few works address them from a more pragmatic point of view. Moreover, the concepts of trust and reputation are often mixed up, which prevents security engineers from designing coherent models where trust and reputation support each other. We believe that counting on this kind of solutions can greatly simplify the task of implementing successful applications that users can trust and are willing to use.

We are aware that learning a framework may be a daunting task. For this reason, we have tried to keep a simple, event-based design. We advocate that the work of learning a new framework can significantly pay off over the burden of hard-coding trust and reputation solutions from scratch every time, in addition to enabling the maintainability of the whole system (34).

There are several issues that require further attention. Up to now, the framework design does not support the implementation of propagation models. Therefore, no trust information can be transferred between entities. For enabling the implementation of these models, it is first required to add algorithms that generate trust chains (i.e. graphs) from the trust relationships stored by the trust server. The developer can define rules to transfer trust between the entities following the trust paths. These rules can be implemented by extending base classes, namely *Concatenator* and *Aggregator*, and by implementing their abstract method *compute()*. The output of the method would be new trust values, which in turn represent new trust relationships between entities that had no prior direct encounters.

A relevant, challenging research question arises: Is the trust framework actually improving the security of the application or improving the decision-making processes? In systems including trust and reputation models, there are usually human users behind the decision-making processes and trust/reputation dynamics. Users usually provide feedback after certain operations and trigger events that update trust relationships and reputation values. It would be interesting to research how the notions of trust and reputation can be effectively tailored for system components and physical devices, without any kind of human interaction. In this direction, we should decide under which terms a system component can trust another component. Even more interestingly, we could ask ourselves whether trust and reputation information can be used to make reconfiguration decisions on the system architecture, or whether system developers can

be provided with usable tools to achieve this. Next chapter addresses these research challenges.

Chapter 5

Enabling Trust and Reputation at Runtime

Preceding chapters discuss how to model and implement trust-aware systems, focusing on design-time solutions. However, two important changes are coming to the Information and Communication Technology (ICT) world that requires looking beyond design time. On the one hand, the service-oriented vision enables on-the-fly improvements upon the functionality available to users. Applications are more dynamic and call for rapid adaptation strategies in order to meet new requirements and to respond to their changing environment. On the other hand, the boundaries between physical and virtual worlds are vanishing with the emergence of the Internet of Things (IoT), where sensors and actuators are embedded in daily life objects and are linked through networks capable of producing vast amount of data. The aforementioned reasons blur boundaries between design and runtime (41) as they prevent designers from envisioning all possible circumstances that might appear during the execution of an application.

Models@run.time is a model-driven approach that supports the runtime adaptation of distributed, heterogeneous systems. It allows working with abstractions and self-adaptive software in order to cope with unforeseeable changes. However, frameworks that accommodate this paradigm have limited support to address security concerns, hindering their usage in real scenarios. We address this challenge by enhancing models@run.time with the notions of trust and reputation, leading to what we call *trust@run.time*.

In this chapter, we take a step ahead and discuss how we can build systems that

make reconfiguration decisions at runtime based on trust relationships and reputation values. With this goal in mind, we present a trust and reputation framework that is integrated into a distributed component-model that implements the models@run.time paradigm, thus allowing the system to include trust in their reasoning process. The framework is illustrated in a chat application by implementing several state-of-the-art trust and reputation models. We show that the framework entails negligible computational overhead and that it requires a minimal amount of work for developers.

The chapter is structured as follows. An introduction to a models@run.time platform called Kevoree is given in Section 5.1, whereas Section 5.2 presents the framework and some details of its implementation. Section 5.3 presents our approach for allowing trust- and reputation-based reconfigurations of the system. An example scenario that illustrates the use of the framework in a chat application is described in Section 5.4. Section 5.5 yields experimental results as for the overhead and the amount of work that the development of such application requires. Finally, Section 5.6 concludes the chapter by presenting research challenges that were identified during the implementation of the framework.

5.1 Kevoree: A Models@run.time Development Platform

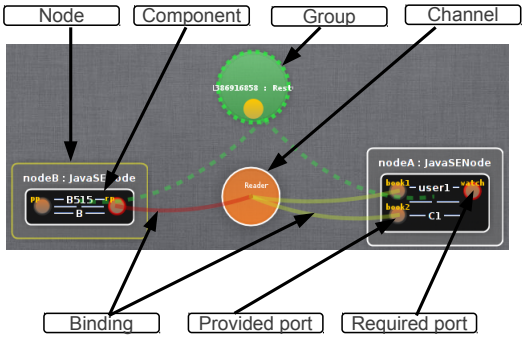
Traditionally, the Model-Driven Software Development area has primarily focused on using models at design, implementation and deployment phases of the SDLC. However, as systems become more adaptable, reconfigurable and self-managing, they are also more prone to failures, which demands putting in place appropriate mechanisms for continuous design and runtime validation and monitoring. Models@run.time (16) aims to tame the complexity of dynamic adaptations by keeping an abstract model of the running system, pushing the idea of reflection one step further. The abstract model is synchronized with the actual system and every change performed on the model is automatically accommodated by the system.

Kevoree[1] is an open-source dynamic component model that relies on *models@run.time* to properly support the design and dynamic adaptation of distributed systems (36). Six concepts constitute the basis of the Kevoree component metamodel. A *node* is an abstraction of a device on which system *components* can be deployed, whereas a *group*

[1] http://kevoree.org

defines a set of nodes that shares the same representation of the reflecting architectural model. A *port* represents an operation that a component provides or requires. A *binding* represents the communication between a port and a *channel*, which allows the communication among components. The core library of Kevoree implements these concepts for several platforms such as Java, Android or Arduino. Figure 5.1 depicts a snapshot of the aforementioned concepts in the Kevoree Editor, which allows building systems in a visual environment.

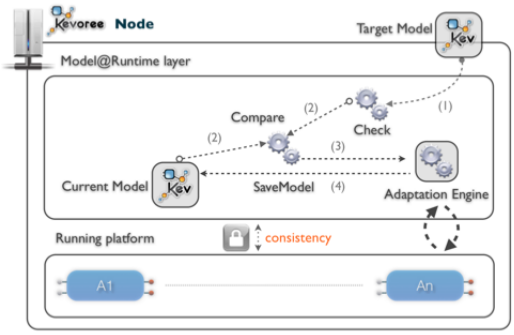
Figure 5.1: Kevoree Architectural Elements



Kevoree adopts the models@run.time paradigm and it boils down the reconfiguration process to moving from one configuration, represented by the current model, to another configuration represented by a target model, as shown in Figure 5.2. First, the target model is checked and validated to ensure a well-formed system configuration. Then, the target model is compared with the current one and this comparison generates an adaptation model that contains a set of abstract primitives that allows the transition from the former to the latter. Finally, the adaptation engine instantiates the primitives to the current platform (e.g. Java) and executes them. If an action fails, the adaptation engine roll backs the configuration to ensure consistency between the models@run.time layer and the running system.

Building an application with Kevoree entails two steps. First, developers create business components through the framework provided by the Kevoree platform. Second, components are deployed on nodes and wired together through bindings and channels. Next sections explain these steps in further detail.

Figure 5.2: Adaptation Process



5.1.1 Kevoree Development Framework

The framework is based on annotations. Components that run on the Kevoree runtime are created by annotating them with *ComponentType*[2].

Components can have *parameters*, which are attributes that are mapped to the reflection layer and can be changed at runtime via *Kevscript*, a scripting language provided by Kevoree, or via the visual editor. Additionally, components provide and require services through their ports.

Listing 5.1 defines a *Console* component with one required port, one provided port and one parameter. The parameter (lines 4-5) determines the appearance of the console frame and can be changed easily at any time both from the editor and with *Kevscript*. The required port (lines 7-8) allows sending text to other consoles, whereas the provided port (lines 10-14) allows receiving text from other consoles and showing it to the user.

Listing 5.1: Definition of Console in Kevoree

```
1 @ComponentType
2 public class Console {
3
4     @Param(defaultValue = "true")
5     protected Boolean showInTab = true;
6
7     @Output
8     protected Port textEntered;
9
```

[2]In the same way, there are annotations to create new channels (*ChannelType*) and nodes (*NodeType*). For the purpose of this chapter, however, we only need to create new components.

```
10      @Input
11      public void showText(Object text)
12      {
13          //Show received text
14      }
15  }
```

The services offered by the Kevoree runtime can be accessed by components through services injected at runtime. Requesting these services entail adding an attribute of the correspondent service type, and annotate such attribute with *@KevoreeInject*. For example, by using the *ModelService* type, developers gain access to the system model and can query it programmatically. Listing 5.2 shows how to find the name of all component instances of a given component type *componentType* running in a node with name *nodeName*[3].

Listing 5.2: Querying the model@runtime layer

```
1   static List<String> getComponentInstanceName(ContainerRoot model,
        String componentType, String nodeName)
2   {
3       List<String> components = new ArrayList<String>();
4       for (ContainerNode node : model.getNodes()) {
5           if (node.getName().equals(nodeName)) {
6               for(ComponentInstance component : node.getComponents()) {
7                   if (component.getTypeDefinition().getName().equals(
                        componentType)) {
8                       components.add(component.getName());
9                   }
10              }
11          }
12      }
13      return components;
14  }
```

First, a list of the existing nodes in the model is retrieved (line 4), and for each of these nodes, we check its name with the searched name. If they are equal, all the components running on the node are retrieved (lines 5-6). For each component, if

[3]Kevoree elements can also be queried using the Kevoree Modeling Framework (`http://kevoree.org/kmf/`), which provides a less verbose and more efficient query language.

the name of the component type matches the searched one, the instance name of the component is added to the result list (lines 7-8), which is finally returned (line 13).

5.1.2 Deployment in Kevoree

Once business components are developed, they can be deployed in nodes and connected through ports. This deployment phase can be realised through the Kevoree editor or by *Kevscript*.

The editor provides a set of basic, built-in libraries (e.g. nodes, basic components and channels) and allows loading custom libraries (i.e. custom business components, customized nodes, channels, etc). It provides drag and drop functionality and a visual representation of the system architecture, as illustrated in Figure 5.3. The models can be converted to *Kevscript* instructions, being possible to save the model as a *.kevs* file containing these instructions.

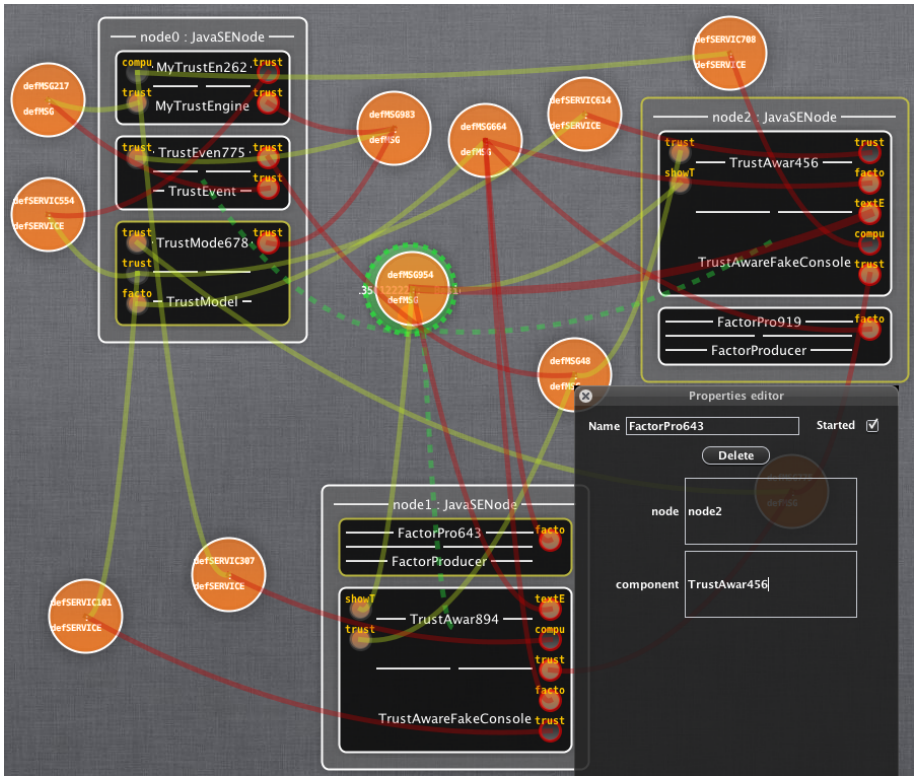
As the complexity of the system increases, the editor may end up overloaded with too much information. In these cases, it is possible to deploy the system by manually specifying *Kevscript* instructions. Figure 5.4 shows an excerpt of this scripting language. In this example, a Java node is added and started, and a component of type *CentralReputationAwareConsole* is deployed on this node and started. The component parameters *showInTab*, *trustContext* and *group* are set.

Kevoree platform does not support reasoning about security concerns, therefore any architectural element such as a node or a software component can join the system without further checks. Also, there is no cross-cutting criteria to guide the runtime changes. Our goal is to provide components with trust and reputation capabilities, which in turn can guide some reconfiguration decisions.

5.2 Integrating Trust and Reputation in Models@Run.time

In this section we explain how we integrate the notions of trust and reputation into the Kevoree component model described in Section 5.1. Developers can use this framework in order to build trust and reputation models for self-adaptive systems. Trust and reputation information that these models generate can be used to make reconfiguration decisions, as further discussed in Section 5.3.

Figure 5.3: Kevoree editor with three nodes



Three components are deployed in *node0* and two in *node1* and *node2*. Components communicate through channels (orange circles) that bind their ports. Parameters can be set for each component (bottom-right grey dialog).

Figure 5.4: Kevscript Instructions

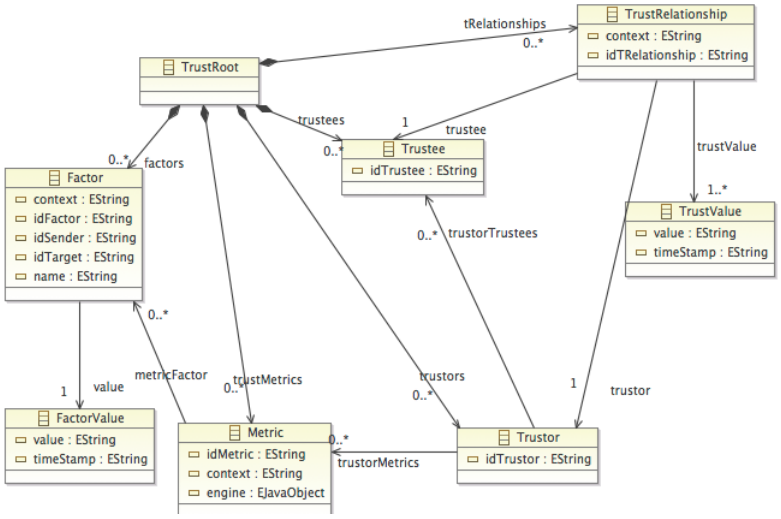
```
add node0 : JavaNode/5.1.3
start node0
add node0.CentralRe452 : CentralReputationAwareConsole/1.0.0
set node0.CentralRe452.showInTab = "true"
set node0.CentralRe452.trustContext = "MyContext"
set node0.CentralRe452.group = "MyGroup"
start node0.CentralRe452
```

The framework consists of an API for developers with some base components that can be extended, some methods that can be overridden, and configuration files. The rest of this section describes the most important aspects of the framework implementation and its integration in the Kevoree component model.

5.2.1 Trust and Reputation Metamodels

We use the Eclipse Modelling Framework (EMF)[4] to create metamodels for trust and reputation. These metamodels gather a set of concepts and relationships among these concepts that abstract away from the particularities of different trust models, in such a way that different metamodels instantiations yield different models. Figures 5.5 and 5.6 show the trust and reputation metamodels, respectively.

Figure 5.5: Trust Metamodel



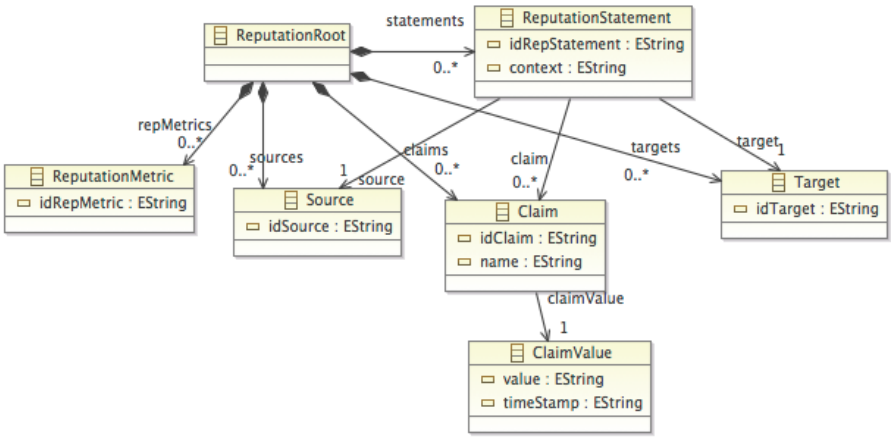
The trust metamodel includes the concept of *TrustRelationship*, which is a tuple of a *Trustor*, *Trustee* and *TrustValue*. *Trustor*s use *Metric*s to evaluate their trust in *Trustee*s. *Metric*s use a set of *Factor*s, which in turn have a *FactorValue*. Different trust models are created by instantiating the entities that play the trustor and trustee roles, the factors that are considered and the way these factors are combined in a metric.

The core concept of a reputation metamodel is a *ReputationStatement*, which is a tuple containing a *Source* entity, a *Target* entity and a *Claim*, which has a *ClaimValue*. A *ReputationMetric* is used in order to aggregate *Claim*s. Reputation models are created by instantiating the entities that play the source and target roles, the way claims are generated and their type, and the way the metric combines the claims.

Note that these metamodels gather many of the concepts explained in Section 2.2.3. In both metamodels, other important concepts from the conceptual framework are in-

[4]http://www.eclipse.org/modeling/emf/

Figure 5.6: Reputation Metamodel



cluded as attributes, like *Context* and *Time*. Other concepts from the conceptual framework that are not presented explicitly in the metamodel are included implicitly in the implementation. For example, factors can be objective and subjective, but the difference is only made at the implementation level with methods available to entities, such as `addSubjectiveFactor`. *Engine*s are concrete implementations of *Metric*s.

Another example concerns centralized and distributed reputation models. As we will see in the next section, centralized reputation models include entities that must send their claims to a component that stores them and which compute reputation, whereas distributed reputation models comprise entities that store their own claims and which compute reputation themselves. In summary, metamodels provide a basic skeleton of relevant concepts, which are enriched during implementation to accommodate more concepts discussed in the conceptual model.

From these metamodels, the EMF generates code that constitutes an API to manage these metamodels. This code does not need to be visible to developers, who can be oblivious about how trust models are instantiated and managed internally by the framework. We use this code as an internal API that acts as an interface between the trust and reputation components offered to developers and the underlying trust or reputation model.

The following sections describe the trust and reputation components, respectively, that constitute the framework.

5.2.2 Trust Framework

This section describes how the trust part of the framework is implemented. As mentioned earlier, this implementation is hidden from developers, as they do not need to know the implementation details in order to use the framework.

One of the main components in the trust framework is *TrustEntity*, which describes an entity capable of participating in a trust relationship. That is, each business component that we want to include in the trust dynamics must inherit from this component. Listing 5.3 shows an excerpt of the implementation.

Listing 5.3: TrustEntity Component

```
1  @ComponentType
2  public class TrustEntity<T, F>
3  {
4      @Param(defaultValue = "both")
5      private String role;
6
7      @Param(defaultValue = "MyContext")
8      private String trustContext;
9
10     @Param(defaultValue = "MyGroup")
11     private String group;
12
13     @Param(defaultValue = "0")
14     private String bootstrappingTrustValue;
15
16     @Param
17     private String subjectiveFactorsFilePath;
18
19      @Output
20     private Port requestTrustUpdate;
21
22     @Output
23     private Port initTrustRelationships;
24
25     @Output
26     private Port addFactor;
27
28    @KevoreeInject
29     private Context context;
```

```
30
31      private String uid;
32
33     @Start
34      protected final void start()
35      {
36           uid = context.getInstanceName() + "@" + context.getNodeName
                 ();
37            initializeTrustRelationships();
38           storeSubjectiveFactors();
39      }
40 }
```

We use Java generics[5] in order to allow developers to set the types for the trust values and the factor values, respectively. We define several parameters. The *role* parameter states whether the entity plays a trustor role, a trustee role, or both roles. The entity can also specify a *trust context* where its relationships are framed. Entities can belong to *group*s and their relationships are to be initialized according to the value of *bootstrappingTrustValue* parameter. The last parameter denotes the name of a file containing *subjective factors* information during initialization.

A trust entity requires services in order to update a trust relationship through the port *requestTrustUpdate*, to initialize its trust relationships through *initTrustRelationships*, and to add factors through *addFactor*. In the *start()* method, which will be called by the Kevoree framework at start-up, a unique identifier for the component is generated by the context service of Kevoree, which provides some basic context information such as the name of the instance and the node where the instance is running. Then, a request to initialize trust relationships is sent to a *TrustManager* component, and finally subjective factors are stored in the model.

Subjective factors are initialized by means of a file that the developer can configure for each trust entity. The format of the file is:

$$FactorName\ FactorValue\ <TargetEntity>$$

The last parameter is optional and denotes the entity to which the subjective factor applies. For instance, if an entity A thinks that another entity B is competent, the file with A's subjective factors would include:

[5] http://docs.oracle.com/javase/tutorial/java/generics/

PerceivedCompetence High B

The most important methods offered by this component are *changeSubjectiveFactor()*, *requestTrustUpdate()*, and *trustRelationshipUpdated()*. The former allows trust entities to increment or decrement an existing subjective factor or to create a new one. The second one requests an update of a trust relationships with a trustee, and the latter is called by the framework when the update is done. Clients can invoke the first two methods and can override the last one in order to make business-level decisions based on trust values. By default, when the client calls *requestTrustUpdate()*, reconfiguration of the system might occur, although developers can inhibit this reconfiguration by invoking an overloaded version of the method. Trust entities also can access recently computed trust values through the method *getLastTrustValue()*, which acts like a cache, saving network resources.

As mentioned above, trust relationships are initialized in the method *initializeTrustRelationships()*, which is depicted in Listing 5.4. This method calls the static method *getTrusteesInstanceName()*, provided by the *GetHelper* class, which is a utility class that allows querying and retrieving information from the reflection layer. We consider that an entity is trustee with respect to another entity if it plays the role *trustee* and it has the same context as the latter. Once we have all the trustees of the entity, we create a trust relationship structure with the context of the relationship, the trustor, the trustee, the initial value of the relationship and a time stamp, and send this structure to the *TrustModel* component. These two components, the *TrustEntity* and *TrustModel*, will be eventually connected through their ports during the deployment phase, as explained in Section 5.1.

Listing 5.4: Trust Entities Initialization

```
1  HashMap<String, List<String>> trusteesAndNodes = GetHelper.
       getTrusteesInstanceName( model.getCurrentModel().getModel(),
       trustContext, trustorInstanceName );
2
3  trustees = new ArrayList<String>();
4
5  //For every node in the model...
6  for (String nodeName : trusteesAndNodes.keySet()) {
7     //...get the list of trustees running on that node
8     for (String compName : trusteesAndNodes.get(nodeName)) {
```

```
9           //The uid of a component is of the form:
                compInstance@nodeWhereRunning
10          trustees.add(compName + "@" + nodeName);
11      }
12  }
13
14  for (String trustee : trustees)
15  {
16      long ts = Calendar.getInstance().getTime().getTime();
17      TrustRelationInfo tri = new TrustRelationInfo( trustContext, uid,
            trustee, bootstrappingTrustValue, ts );
18      initTrustRelationships.send( tri );
19  }
```

The *TrustModel* component manages the trust metamodel and is in charge of computing trust values. Listing 5.5 shows how the component adds the trust relationship to the metamodel, for which the EMF API (see Section 5.2.1) is used.

Listing 5.5: Adding Trust Relationships with the EMF API

```
1  private void addTrustRelationship( String context, String idTrustor,
       String idTrustee, String initialValue, long timeStamp )
2  {
3
4      Trustee trustee = trustModel.findTrusteesByID( idTrustee );
5      if ( trustee == null )
6      {
7          trustee = factory.createTrustee();
8          trustee.setIdTrustee( idTrustee );
9          trustModel.addTrustees( trustee );
10     }
11
12     //Creation of the rest of trust relationships elements
13 }
```

Clients of *TrustModel* can invoke several methods in order to retrieve factor information, and can override two methods, *compute()* and *computeThreshold()*. This is illustrated in Listing 5.6. Retrieving factor values is essential in order to implement the trust engines, which need these values to compute trust. Trust engines are implemented by overriding the *compute()* method, and optionally, the *computeThreshold()* method.

The last important component of the trust framework is the *FactorProducer*[6]. This type of entity adds objective factors about other entities by using low-level platform services that provide information about the components and their communications. This is a key component to QoS-based trust models, as it allows the model to easily take into account information about the response times, number of failures, uptime percentage of services, and so on.

Listing 5.6: TrustModel Component

```
1  protected final String getFactorValue( String context, String name,
       String uidTarget )
2  {
3     for( Factor f : trustModel.getFactors() )
4     {
5        if (context.equals(f.getContext()) && name.equals(f.getName())
             && uidTarget.equals(f.getIdTarget()))
6        {
7              return f.getValue().getValue();
8           }
9     }
10    return null;
11 }
12
13 public T compute(String context, String idTrustee, String idTrustor)
14 {
15    return null;
16 }
17
18 protected T computeThreshold(String context, String idTrustee,
       String idTrustor)
19 {
20    return null;
21 }
```

Clients of this component must set the instance identifier of the target entity and override the method *doEvaluation()*. This component can work in two ways: by assigning a value at initialization time, and by monitoring the target at a regular interval that developers can also specify in another parameter. The value returned by the method is

[6]In the scope of this dissertation, we are interested in the social notions of trust and reputation, therefore the chosen models in Section 5.4 do not use this component.

automatically included as a factor in the method, and engines can retrieve the factor during the computation.

The next section revises the most important implementation details of the reputation framework.

5.2.3 Reputation Framework

The reputation framework allows the implementation of centralized and distributed reputation models by means of *CentralReputableEntity* and *DistReputableEntity* components, respectively. The former requires the communication with a *ReputationManager* component in order to send to it the claims and retrieve reputation information, whereas the latter requires a *ReputationEngine* that will be in charge of computing reputation for the component.

An excerpt of the *CentralReputableEntity* implementation is presented in Listing 5.7. Again, reputation takes place in a *trust context*, and entities may belong to a *group* and need a unique identifier *uid*. These entities also require a port through which to send their claims and request reputation information. Two important methods offered to clients are *makeClaim()* and *requestReputation()*, which allow sending claims to and retrieving reputation information from the *ReputationManager*. Another important method that client code can override is *reputationReceived()*, as depicted in Listing 5.7. This method is called by the framework when a new reputation value of an entity is computed. When a reputation update is requested, the default behaviour is reconfiguring the system in case it is required, although the client can explicitly disable the reconfiguration by setting the corresponding parameter to *false*. *CentralReputableEntity* also caches the last computed reputation values in order to provide fast access, through the method *getLastReputation()*, to the reputation of an entity with which it interacted in the past.

Listing 5.7: CentralReputableEntity Component

```
1 @ComponentType
2 public class CentralReputableEntity<T> implements IClaimSource {
3
4     @Param(defaultValue = "MyContext")
5     private String trustContext;
6
7     @Param(defaultValue = "MyGroup")
```

```
8       private String group;
9
10      @Output
11      private Port sendClaim;
12
13      @Output
14      private Port requestReputation;
15
16      private String uid;
17
18      protected void reputationReceived( String target, T newVal ) { }
19  }
```

The other important component of a centralized reputation model is the *ReputationManager*, which interacts with the reputation metamodel and offers methods to retrieve claims information. It also provides the method *compute()*, which clients must implement for their reputation engines. An excerpt of the implementation is depicted in Listing 5.8. As an example, the method *getClaimValues()* retrieve the values of all the claims that are issued in a specific *context*, with a specific *name* and for a concrete *target* entity. Clients can inherit from this class and override the method *compute()*.

Listing 5.8: ReputationManager Component

```
1  @ComponentType
2  public class ReputationManager<T> implements IComputationEngine
3  {
4     //...
5
6     protected final List<String> getClaimsValues( String context,
          String name, String target ) {
7           List<String> claims = new ArrayList<String>();
8           for (ReputationStatement rs : repRoot.getStatements())
9           {
10              if (rs.getContext().equals( context ) && rs.getTarget().
                    getIdTarget().equals( target ))
11              {
12                  for(Claim claim : rs.getClaim())
13                  {
14                      if(claim.getName().equals(name))
15                      {
```

```
16                          claims.add( claim.getClaimValue().getValue()
                                );
17                      }
18                  }
19              }
20          }
21          return claims;
22      }
23
24      public T compute( String context, String idTarget, String
            idSource )
25      {
26          return null;
27      }
28
29      //...
30  }
```

The other type of reputation model, namely distributed reputation models, are built around two components: *DistReputableEntity* and *ReputationEngine*. The former represents an entity capable of issuing claims (such as *CentralReputableEntity*), but which is responsible to store its own claims and to send them to other entities that may request them. The latter is a reputation engine that belongs to a distributed entity. A reputation engine is bound to an entity at start-up, as illustrated in Listing 5.9. As in the case of the *CentralReputableEntity*, this component provides several methods to issue claims and to request reputation updates. Clients can also override methods to react when a new reputation value arrives.

Listing 5.9: DistReputableEntity Component and Reputation Engine Initialization

```
1   public class DistReputableEntity<T> implements IClaimSource {
2
3       @Param(defaultValue = "MyContext")
4       private String trustContext;
5
6       @Param(defaultValue = "MyValue")
7       private String group;
8
9       @KevoreeInject
10      private Context ctx;
```

```
11
12      @Output
13      private Port requestClaim;
14
15      private String uid;
16      private ReputationEngine<T> reputationEngine;
17
18      @Start
19      public void start( ReputationEngine repEngine )
20      {
21          uid = ctx.getInstanceName() + "@" + ctx.getNodeName();
22          reputationEngine = repEngine;
23
24          //...
25
26      }
27
28      //...
29  }
```

Listing 5.10: ReputationEngine Class

```
1  protected final List<ReputationStatementInfo> getClaimsFromSource(
       String name, String idSource)
2  {
3          List<ReputationStatementInfo> claims=new ArrayList();
4          for( ReputationStatementInfo cInfo : reputationStatements )
5          {
6              if( name.equals(cInfo.getClaim().getName()) && cInfo.
                   getSource().equals( idSource ))
7              {
8                  claims.add( cInfo );
9              }
10         }
11         return claims;
12     }
```

Clients using *ReputationEngine* must implement the method *compute()* and can gain access to the claims in the system by methods like the one shown in Listing 5.10, which retrieves all the claims made by an entity about a particular subject, represented by the name of the claim. In order to have access to all the claims, the entities send

them all to their engines right before the computation.

Next section discusses how the framework provides trust-based self-adaptation.

5.3 Trust-based Self-Adaptation

The interesting advantage of implementing the framework on top of a self-adaptive platform is that developers can use trust and reputation information to change the system at runtime. Regardless of the implemented model, the output of the model (i.e. the trust or reputation value) can be used to make reconfiguration decisions.

5.3.1 Policy-based Reconfiguration

The framework supports this reconfiguration process by means of policies in the form of simple rules. In the case of reputation-based reconfiguration, rules are in the following form:

ComponentType BooleanCondition Action Arguments

where *ComponentType* is the type of the component for which the reputation is to be used in the *BooleanCondition*. If it is true, then *Action* is executed with the required *Arguments*. Trust-based reconfiguration rules are similar:

ComponentType ComponentType BooleanCondition/"threshold" Action Arguments

where the first *ComponentType* is the type of the trustor and the second one is the type of the trustee. Either the trust value is compared according to the *BooleanCondition* or the model threshold is used to determine whether the *Action* should be executed with the required *Arguments*.

The actions currently implemented in the framework are *remove* and *substitute*. The former does not require arguments and tells the runtime system to remove the component if the boolean condition is met. The latter requires at least one argument, which is another component type, and tells the system to substitute the current component for another component of the new type if the condition is fulfilled.

Let us illustrate with a couple of examples. Consider the following reputation-based reconfiguration policy file:

CentralReputationAwareConsole <1 substitute FilteredCentralReputationAwareConsole
FilteredCentralReputationAwareConsole <0 remove

This policy is specifying the following: "if the reputation of any instance of type *CentralReputationAwareConsole* is less than 1, then substitute it for another component instance of type *FilteredCentralReputationAwareConsole*. Likewise, if the reputation of any instance of type *FilteredCentralReputationAwareConsole* is less than 0, then remove it".

Consider now the following trust-based reconfiguration policy file:

TrustAwareConsole TrustAwareConsole threshold substitute TrustAwareConsole
TrustAwareConsole TrustAwareConsole <0 remove

The policy is specifying the following: "if a trustor of type *TrustAwareConsole* does not trust a trustee of type *TrustAwareConsole* over a threshold (defined by the model), then substitute the trustee for a new component of type *TrustAwareConsole*. Likewise, if the trust that a trustor of type *TrustAwareConsole* places in a trustee of type *TrustAwareConsole* is less than 0, then remove the trustee".

In addition to the new component type, the *substitute* action can have an undefined number of parameters that represent attributes of the new instances and their values. If no parameters are found, it is assumed that new instances should replicate the same values of the attributes of the instances removed.

As an example, consider the following:

Console <1 substitute FilteredConsole group A
FilteredConsole >5 substitute Console

This policy is specifying the following: "if the reputation of any instance of type *Console* is less than 1, then substitute it for another component instance of type *FilteredConsole* and set its parameter *group* to the value *A*. Likewise, if the reputation of any instance of type *FilteredConsole* is more than 5, then substitute it for a component instance of type *Console* and set all the parameters that have the same name to the same values that the previous instance had".

5.3.2 Implementation

In this section, we explain some details of the implementation. The main class is *ScriptEngine*, which encapsulates the actions and generates the *Kevscript* instructions. The reputation framework provides the class *ReputationRulesEngine*, which processes the policy file and calls the script engine to generate the adaptation script. Likewise, the trust framework uses the class *TrustRulesEngine* for the same purpose. Listing 5.11 shows the initialization of *DistReputableEntity*. Note that instances of *ScriptEngine* and *ReputationRulesEngine* are created, and that the former is passed as an argument to the latter, together with the name of the policy file (*RepRules.policy* by default). The listing also shows an internal method of the framework which calls the *compute()* method of the reputation engine and which determines whether the user wants to reconfigure the system, in which case the method *executeRules()* of the *ReputationRulesEngine* class is called. In turn, this method reads the file and executes, via the instance of *ScriptEngine*, the rules for which the boolean conditions are met.

Listing 5.11: Reconfiguration Rules Processing

```
1  @ComponentType
2  public class DistReputableEntity<T> implements IClaimSource {
3
4      @Start
5       public void start( ReputationEngine repEngine, String fileName )
6       {
7           se = new ScriptEngine( model );
8           rre = new ReputationRulesEngine( fileName, se );
9       }
10
11     private void computeReputation( String idTarget, boolean
           reconfigure, List<ReputationStatementInfo> rsInfo )
12      {
13          T res = reputationEngine.compute( model, trustContext,
                idTarget, uid, rsInfo, this );
14
15          if ( reconfigure )
16          {
17              rre.executeRules( model, idTarget, res.toString() );
18          }
19
```

```
20            //...
21        }
22    }
```

As an example of how *ScriptEngine* executes *Kevscript* instructions, Listing 5.12 shows the implementation of the action *remove*. Once the name of the instance in the reflection layer is obtained (by trimming the node name where it is executing), the script is executed by means of the *ModelService* variable *model*, which allows submitting scripts as a *String* and checking the results in a callback.

Listing 5.12: ScriptEngine: Remove Component

```
1  private void removeComponent( String idComponent )
2  {
3      StringTokenizer st = new StringTokenizer( idComponent, "@" );
4      String instance = st.nextToken();
5      String node = st.nextToken();
6
7       String script = "remove " + node + "." + instance;
8       model.submitScript(script, new UpdateCallback() {
9       @Override
10       public void run(Boolean applied) {
11        //check if applied = true
12       }
13     });
14 }
```

Substituting a component entails more work, as a new component must be created and must be ensured that it will be able to continue its communication with the rest of components. Given that the code is much longer, we simply enumerate the steps required for this action[7]:

1. Obtain information (if necessary) about the attributes of the instance to be removed.

2. Obtain information about the bindings and channels of the component to be removed.

[7]The interested reader can check the Javadoc and source code in `https://www.nics.uma.es/development/trust-and-reputation-framework-modelsruntime`

3. Remove component.
4. Create new instance name of the type specified in the policy file.
5. Add the component and link it to the channels via new bindings.
6. Add new attributes (which could be the same as the attributes of the just removed instance).

5.4 Application Example: A Trust-Aware Distributed Chat

In this section, we explain how we can implement several well-known trust and reputation models. First, we provide a brief description of each model followed by high-level steps required to implement them. We also show the most relevant code for each model[8].

The chosen scenario is a distributed chat application, because it is a simple scenario that allows illustrating the use of trust and reputation models easily. The mechanics are similar for every model: a console receives a message from another console and inspects the contents of the message. Depending on these contents (e.g. if it detects a swear word), it provides a stimuli to the trust or reputation model. This stimuli may come in the form of claims or changes in factors, as it will be illustrated in each model.

5.4.1 eBay Model

In the eBay reputation model, after a transaction finishes, both the seller and the buyer can rate each other by a positive (-1), neutral (0) or negative feedback (1). The reputation for an individual is then calculated by summing up the distinct ratings for such individual (108). The model is centralized, in such a way that all the feedbacks are sent to a central system that computes the reputation, and each user can query and see this reputation in the form of an html page.

This model is mapped to our example in the following way. Once a console receives a message from another console, it looks for offensive words contained in the message, which are previously configured by the user. If any offensive word is included, then a negative feedback about the sender is issued (-1); otherwise, a positive feedback is submitted (1).

[8]We omit some error checking and boiler plate code for the sake of better understandability.

The implementation of this model in the framework requires the following coarse-grained steps:

- Consoles must inherit from *CentralReputableEntity*,
- Consoles invoke the method *makeClaim* with the appropriate value upon receiving a message.
- Reputation engine inherits from *ReputationManager* and overrides the method *compute()*.

Further details are provided in Listing 5.13. In the former, a console component inheriting from a central reputable entity is created. The generic type is instantiated to *Integer* because that is the format in which we want to represent the reputation value. Then, upon receiving a message through the input port *showText*, it retrieves the identity of the sender and the text itself (provided by the console or by the middleware itself), and if any offensive word is found, it issues a negative claim with name *CleanWords*; otherwise, it issues a positive one. The message is temporarily stored in *lastMessageReceived* (if there are no offensive words) and an update of the reputation of the sender is requested, which will call the reputation engine. In turn, the reputation engine will automatically call the method *reputationReceived()* with the identity of the entity and the new reputation value, and we could perform additional checks to determine whether the console should print the message or not.

Listing 5.13: Console in the Ebay Reputation Model

```
1  @ComponentType
2  public class CentralReputationAwareConsole extends
       CentralReputableEntity<Integer> {
3
4      //It stores the last message receive
5      private String lastMessageReceived;
6
7      @Input
8       public void showText(Object text)
9       {
10          if( text != null ) {
11              String msg = text.toString();
12              StringTokenizer st = new StringTokenizer( msg, "." );
```

```
13              String idTarget = st.nextToken();
14              String message = st.nextToken();
15              if( badWordsInMessage( message ) )
16              {
17                  lastMessageReceived = "";
18                  makeClaim("CleanWords", "-1", idTarget);
19              }
20              else
21              {
22                   lastMessageReceived = message;
23                   makeClaim( "CleanWords", "1", idTarget );
24              }
25              requestReputation( idTarget );
26          }
27      }
28
29      @Override
30      public void reputationReceived( String target, Integer newVal )
31      {
32        //We could check if the reputation is above a given threshold
              prior to showing the message
33          thisConsole.appendIncomming( "->" + lastMessageReceived );
34      }
35
36  }
```

The reputation model, depicted in Listing 5.14, implements the Ebay reputation engine. First, it retrieves all the claims named *CleanWords* about the target *idTarget*. If there are no claims about the target, then a default value is returned, otherwise, the reputation value is computed by summing up all the claim values.

Listing 5.14: Ebay Reputation Engine

```
1  @ComponentType
2  public final class EBayReputationModel extends ReputationManager<
       Integer> {
3
4      @Override
5      public Integer compute( String context, String idTarget, String
           idSource ) {
6
```

```
7           List<String> claims = getClaimsValues( context, "CleanWords"
                , idTarget );
8           int res = 0;
9           if ( claims != null )
10          {
11           //By default reputation
12              if( claims.size() == 0 )
13              {
14                  return 1; //Initial reputation
15              }
16              for ( String c : claims )
17              {
18                  res += Integer.valueOf( c );
19              }
20          }
21          return res;
22      }
23  }
```

5.4.2 Marsh's Trust Model

Marsh was one of the first authors that formalised trust in a computational setting (75). His model considers the following factors:

- Utility (U_x): this factor measures the utility that entities would obtain from a successful collaboration.
- Basic trust or trust disposition (T_x): this subjective factor indicates what is the attitude of an entity towards higher or lower values of trust.
- Importance (I_x): this subjective factor indicates how important a situation is for an entity.
- Perceived Competence: this subjective factor states how competent the trustor thinks that the trustee is for the task in play.
- Perceived Risk: this subjective factor denotes how risky the entity thinks the situation is.

- General trust ($T_x(y)$): this refers to the trust that the trustor places in the trustee as a consequence of the history of interactions.

The model uses the aforementioned factors to calculate the so-called *situational trust*, which according to the author is the most important when considering trust in cooperative situations. In particular, situational trust is defined as:

$$T_x(y,\alpha) = U_x(\alpha) \times I_x(\alpha) \times \widehat{T_x(y)}$$

where x is the trustor, y is the trustee, and α is the situation[9]. Marsh also models what he calls a *cooperation threshold*:

$$CT_x(\alpha) = \frac{PerceivedRisk_x(\alpha)}{PerceivedCompetence_x(y,\alpha) + \widehat{T_x(y)}} \times I_x(\alpha)$$

The model states that an agent engages in a collaboration with another agent if the situational trust is greater than the cooperation threshold. In order to implement the model in the framework, the following steps must be performed:

- Consoles inherit from *TrustEntity*.
- Consoles add their subjective factors.
- Consoles change their factors in response to the received messages.
- Trust engines inherit from *TrustModel* and overrides the methods *compute()* and *computeThreshold()*.

In order to initialize their subjective factors, we create a simple text file with a list of $factor\ value < target >$, where $factor$ represents the factor name, *value* denotes the value of the factor and *target*, which is an optional parameter, the name of the component instance to which the factor refers. An example of this file for one of the consoles in Marsh's is illustrated in Table 5.1. The name of this file is assigned in *Kevscript* or from the editor for each console. Right after the trust relationships of the console have been initialized, the file is read and the factors are stored.

Listing 5.15 shows the console implementation. First, it inherits from *TrustEntity*, the generics of which are instantiated to String and Float, because this is the format

[9]The situation for Marsh is what we call context.

Table 5.1: Trust Factors for Marsh's Model

Factor	Value	Target
utility	1.0	-
trustDisposition	0.8	-
importance	0.4	-
perceivedRisk	0.2	-
perceivedCompetence	0.9	console234@node0

we are computing the trust values and the trust factors, respectively. When a console receives a new message and after retrieving metadata (e.g. sender, which is the trustee of the relationship), it stores the message and looks for offensive words in the text. If any offensive word is found, it invokes the method *changeSubjectiveFactor()*. The arguments tell that the factor *perceivedCompetence* that refers to the *trustee* entity should be decreased by 0.1, and in case this factor does not exist, it should be initialized to 0.5[10]. Finally, the console requests a trust update about its trustee.

When the trust update is completed by the trust engine, the method *trustRelationshipUpdated()* is called, which indicates the trustee to which it refers, and a list with two potential values. The first value represents the actual new trust value, whereas the second value is the threshold value[11]. Depending on their relationship, the message is finally printed or not.

Listing 5.15: Console in Marsh Trust Model

```
1  @ComponentType
2  public class TrustAwareConsole extends TrustEntity<String,Float>
3
4      //...
5
6      @Input
7       public void showText(Object text)
8       {
9           if( text != null ) {
10              String msg = text.toString();
```

[10]The factor may not exist if a reconfiguration has taken place and a new component has been added for which there is not such factor.

[11]We say potentially because not all trust models include a trust threshold computation and therefore, in that case, it would be up to the developer to hard-code a reasonable threshold.

```
11              StringTokenizer st = new StringTokenizer( msg, "." );
12              String trustee = st.nextToken();
13              String group = st.nextToken();
14              String message = st.nextToken();
15              lastMessageReceived.put( trustee, message );
16
17              if( badWordsInMessage( message ))
18              {
19                  changeSubjectiveFactor("perceivedCompetence", -0.1f,
                        0.5f, trustee );
20              }
21              requestTrustUpdate( trustee );
22          }
23      }
24
25      @Override
26      protected void trustRelationshipUpdated(final String trustee,
            List<String> newVal) {
27
28          float trustValue = Float.valueOf(newVal.get(0));
29          float threshold = Float.valueOf(newVal.get(1));
30
31          if( trustValue >= threshold )
32          {
33              thisConsole.appendIncomming( lastMessageReceived.get(
                    trustee ) );
34          }
35      }
36  }
```

The trust engine is shown in Listing 5.16. First, it inherits from *TrustModel* and instantiates the generics to String and Float, which again are the formats of the trust values and trust factors. Developers must implement the methods *compute()* and *incrementFactor()* and can implement the method *computeThreshold()*. The former computes a trust value from the trust factors as discussed earlier in the description of the model. The second method determines how an increment/decrement should be performed depending on the concrete generics instantiation. The latter allows computing a threshold from the trust factors as discussed earlier in the description of the model. Listing 5.16 shows an excerpt of the implementation, in particular the *compute()* and

the *incrementFactor()* methods.

Listing 5.16: Trust Engine for Marsh's Model

```
1  @ComponentType
2  public class MarshModel extends TrustModel<String, Float>
3  {
4      @Override
5      public String compute(String context, String idTrustee, String
           idTrustor)
6      {
7          float utility = Float.valueOf( getFactorValue( context, "
               utility", idTrustor ));
8          float importance = Float.valueOf( getFactorValue(context, "
               importance", idTrustor ));
9          String generalTrustString = getFactorValue( context, "
               generalTrust", idTrustor, idTrustee );
10         float generalTrust = Float.valueOf( getFactorValue( context,
               "generalTrust", idTrustor, idTrustee ));
11
12         float situationalTrust = utility * importance *
13          generalTrust;
14
15         return String.valueOf(situationalTrust);
16     }
17
18     //... Compute threshold in a similar way...
19
20     @Override
21     protected String incrementFactor(String currentValue, String
           increment) {
22         return String.valueOf( Float.valueOf( currentValue ) + Float
               .valueOf( increment ));
23     }
24 }
```

5.4.3 PeerTrust

Xiong and Liu (137) propose PeerTrust, a distributed reputation model oriented towards Peer-to-Peer scenarios. This model, as in the case of most reputation models, builds a reputation score upon feedbacks that peers yield after their collaboration. In addition

to the feedbacks, the model proposes using the following factors:

- Number of transactions that a peer has had with another peer.
- Credibility of the feedback; feedbacks from more trustworthy peers should weight more in the calculation.
- Transaction context, which refers to metadata about the context where the transaction or collaboration is taking place.
- Community context, which relates to incentives for providing feedbacks.

The general trust metric is the following:

$$T(u) = \alpha \cdot \sum_{i=1}^{I(u)} S(u,i) \cdot Cr(p(u,i)) \cdot TF(u,i) + \beta \cdot CF(u)$$

where $I(u)$ is the total number of transactions that peer u had with the rest of peers, $p(u,i)$ denotes the other participating peer in peer u's ith transaction, $S(u,i)$ is the satisfaction peer u receives from $p(u,i)$, $Cr(v)$ denotes the credibility of the feedback submitted by v, $TF(u,i)$ is the transaction context factor for u's ith transaction, and $CF(u)$ denotes the community context factor for peer u. α and β are weights for the collective evaluation and the community context factor.

In our example, we identify each transaction with a message sent and received by two communicating consoles. We lay the community context aside (i.e. $\beta = 0$) and focus entirely on the collective evaluation (i.e. $\alpha = 1$). The authors of the model provide hints about how to calculate the credibility and the context factor. In particular, the credibility can be calculated using the following formula:

$$Cr(u) = \frac{T(p(u,i))}{\sum_{i=1}^{I(u)} T(p(u,i))}$$

which uses the ratio between the current reputation of the peer that sent the feedback and the total reputation of all peers that previously had a collaboration with u.

Regarding the context factor, the authors mention that a time-stamp of the transaction can be used in order to give more relevance to more recent transactions. One way to model this is by the following formula:

$$TF(u,i) = \frac{TS(u,i)}{CurrentTime}$$

where $TS(u,i)$ is the time when the ith transaction took place.

Once we have this, we can implement the model in the framework following these high-level steps:

- Consoles must inherit from *DistReputableEntity*,
- Consoles invoke the method *makeClaim* upon receiving a message.
- Reputation engine inherits from *ReputationEngine* and overrides the method *compute*.
- Console is assigned the reputation engine created.

The console code is similar to the one shown in Listing 5.13. The difference is that in this case we are dealing with a distributed reputation model, therefore consoles must inherit from *DistKevReputableEntity* and each console is responsible to compute reputation values, instead of delegating this task to a reputation manager. Thus, in the start method of the console, we need to specify which reputation engine the console will use, as depicted in Listing 5.17.

Listing 5.17: Binding Reputation Engine and Console

```
1  @ComponentType
2  public class DistReputationAwareConsole extends
       DistKevReputableEntity<Float>
3
4       @Start
5      public void startConsole()
6      {
7          super.start( new PeerTrustModel() );
8
9          //More console-specific initialization stuff
10     }
11 }
```

The code for the reputation engine is illustrated in Listing 5.18, which implements the formula described earlier for PeerTrust. Note that the method *calculateTotalReputation()* is not part of the framework, but a way to modularize the *compute* method.

Listing 5.18: Reputation Engine for PeerTrust

```
1  public class PeerTrustModel extends ReputationEngine<Float>
2  {
3      @Override
4      public Float compute(String context, String idTarget, String
           idSource)
5      {
6          List<ReputationStatementInfo> allStatementsAboutTarget =
               getClaimsAboutTarget( "CleanWords", idTarget );
7          float totalReputation = calculateTotalReputation(idTarget);
8          double currentTime = (double) Calendar.getInstance().getTime
               ().getTime();
9          float targetReputation = 0.0f;
10         for (ReputationStatementInfo rs:allStatementsAboutTarget)
11         {
12             String source = rs.getSource();
13             float sourceReputation = Float.valueOf(
                   getLastReputation(source) );
14             float credibility = sourceReputation / totalReputation;
15             targetReputation += Float.valueOf( rs.getClaim().
                   getValue() ) * credibility * Double.valueOf( rs.
                   getTimeStamp() ) / currentTime;
16         }
17         return targetReputation;
18     }
19
20     private float calculateTotalReputation(String idTarget)
21     {
22         List<ReputationStatementInfo> allStatementsAboutTarget =
               getClaimsAboutTarget( "CleanWords", idTarget );
23         Set<String> consideredEntities = new HashSet();
24         float totalRep = 0.0f;
25         for (ReputationStatementInfo rs:allStatementsAboutTarget)
26         {
27             String sourceEntity = rs.getSource();
28             String sourceReputationString = getLastReputation(
                   sourceEntity );
29             float sourceReputation = Float.valueOf(
                   getLastReputation( sourceEntity ) );
30             consideredEntities.add( sourceEntity );
31             totalRep += sourceReputation;
```

```
32          }
33          float targetReputation = Float.valueOf( getLastReputation(
                idTarget ));
34          totalRep += targetReputation;
35          return totalRep;
36      }
37  }
```

5.4.4 REGRET

REGRET (114) is a trust model that considers three reputation values, one for each considered dimension: an individual dimension, a social dimension and an ontological dimension. The three reputation values are calculated from a set of impressions gathered by the entities. These impressions are about a subject and a target, and map to what we call claims.

The individual dimension calculates a so-called subjective reputation value by using impressions of an agent about another target agent, as follows:

$$R_{a \to b}(subject) = \sum \rho(t, t_i) \cdot W_i$$

where a is the source entity, b is the target entity, W_i is the claim value in the range $[-1, 1]$, and ρ is a function that gives recent impressions a higher weight.

For the social dimension, the model considers that agents belong to groups, denoted by $\mathcal{A}$, $\mathcal{B}$, etc, and calculates the reputation at the group level, considering the impressions about each agent of the group. In particular, the model considers:

$$R_{a \to \mathcal{B}}(subject) = \sum_{b_i \in \mathcal{B}} w \cdot R_{a \to b_i}(subject)$$
$$R_{\mathcal{A} \to b}(subject) = \sum_{a_i \in \mathcal{A}} w \cdot R_{a_i \to b}(subject)$$
$$R_{\mathcal{A} \to \mathcal{B}}(subject) = \sum_{a_i \in \mathcal{A}} w \cdot R_{a_i \to \mathcal{B}}(subject)$$

where w are weights that must sum up 1. The final reputation value consists of a weighted sum of all the previous values.

The model also considers an ontological dimension, where a subject (or context) might be decomposed into other subjects, which allows generalizing a reputation value for a new subject from weighting the contributing existing subjects.

In order to implement this model in our framework, we make some slight simplifications. The most important one is that we do not consider the ontological dimension, because contexts relationships are not supported in the framework. Also, in order to simplify calculations and show a clearer code, we assume a uniform distribution of weights across impressions and we do not consider reliability of the reputation values.

The coarse-grained steps we must follow for the implementation of this model are the following:

- Inheriting from *DistKevReputableEntity*, invoking the method *makeClaim* upon receiving a message, which simulates the impressions.
- Set the group to which each entity belongs.
- The reputation engine must retrieve the impressions of all entities to compute the different reputation values, and the groups to which each entity belongs.

The code for the console is the same as the one depicted in Listing 5.17, except that we need to bind the console to another reputation engine in the start method. The reputation engine simply implements the formula described previously, as shown in Listing 5.19.

Listing 5.19: Reputation Engine for REGRET

```
1  public class RegretReputationModel extends ReputationEngine<Float>
2  {
3        @Override
4      public Float compute(String context, String idTarget, String
           idSource) {
5
6          //1) Calculate subjective reputation
7          List<ReputationStatementInfo> claims = getClaims( "
               CleanWords", idSource, idTarget );
8          float subjectiveReputation = 0.0f;
9          float totalClaims = claims.size();
10         double currentTime = (double) Calendar.getInstance().getTime
               ().getTime();
11         for( ReputationStatementInfo rs : claims )
12         {
13             float claimVal = Float.valueOf( rs.getClaim().getValue()
                   );
```

```
14              double claimTimeStamp = (double) rs.getTimeStamp();
15              subjectiveReputation += (claimVal / totalClaims) * (
                    claimTimeStamp / currentTime);
16          }
17
18          //2) Now, retrieve all the claims that idSource made about
                any entity in the same group as idTarget
19          String groupTarget = getParam( idTarget, "group" );
20          List<ReputationStatementInfo> targetGroupClaims =
                getClaimsFromSource("CleanWords", idSource);
21          float targetGroupReputation = 0.0f;
22          currentTime =  Calendar.getInstance().getTime().getTime();
23          for( ReputationStatementInfo rs: targetGroupClaims )
24          {
25              //We don't want to consider claims about the target
                    itself
26              if ( !idTarget.equals( rs.getTarget() ))
27              {
28                  String g = getParam(rs.getTarget(), "group");
29                  if (groupTarget.equals(g)) {
30                      float claimVal = Float.valueOf(rs.getClaim().
                            getValue());
31                      double claimTimeStamp = (double) rs.getTimeStamp
                            ();
32                      targetGroupReputation += (claimVal / totalClaims
                            ) * (claimTimeStamp / currentTime);
33                  }
34              }
35          }
36
37          //3) Now, retrieve all the claims that any entity in the
                same group as idSource made about idTarget
38          String groupSource = getParam( idSource, "group" );
39          List<ReputationStatementInfo> sourceGroupClaims =
                getClaimsAboutTarget( "CleanWords", idTarget );
40          float sourceGroupReputation = 0.0f;
41          currentTime =  Calendar.getInstance().getTime().getTime();
42          for( ReputationStatementInfo rs : sourceGroupClaims )
43          {
44              //We don't want to consider claims from the source
                    itself
```

```
45              if ( !idSource.equals( rs.getSource() ))
46              {
47                  String g = getParam(rs.getSource(), "group");
48                  if (groupSource.equals(g)) {
49                      float claimVal = Float.valueOf( rs.getClaim().
                            getValue() );
50                      double claimTimeStamp = (double) rs.getTimeStamp
                            ();
51                      sourceGroupReputation += (claimVal / totalClaims
                            ) * (claimTimeStamp / currentTime);
52                  }
53              }
54          }
55
56          return new Float( subjectiveReputation +
                targetGroupReputation + sourceGroupReputation );
57  }
```

5.5 Experimenal Results

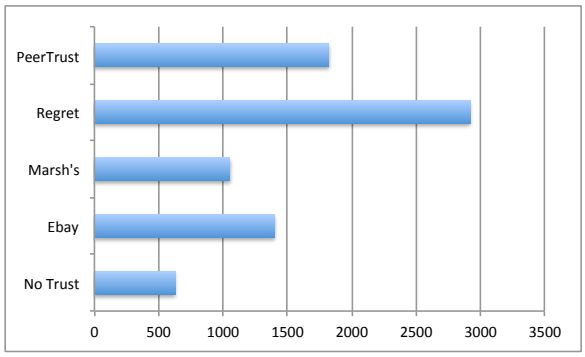
This section explains the experiment that we carry out in order to measure the performance overhead that the framework entails, as well as the amount of work that developers need to invest during the implementation of the models.

The application used for the experiment is the one explained in Section 5.4. In order to ignore network latency, both consoles are executed on the same platform, which is a 2010 Macbook Pro Intel Core 2 Duo, with 4GB 1067 MHz DDR3 RAM.

The experiment is as follows. First, we measure the time elapsed between the time the first console sends a message and the second console shows it, without any trust or reputation involved. Then, for each trust or reputation model considered in Section 5.4, we measure this same time. In order to account for the computation engines, the receiver console shows the text only after it has received an update of the trust or reputation of the sender console. Each measure is actually an average of 100 individual measures to provide more statistically meaningful results, which are depicted in Figure 5.7.

As the figure depicts, there is a small overhead when using the framewok, although this overhead comes in terms of microseconds. The least overhead comes from Marsh's

Figure 5.7: Execution Time (measured in microseconds)



model, whereas the greatest comes from REGRET, something expected given its more complex computation engine.

The amount of work that takes for developers to implement the models is similar for all the models, as depicted in Table 5.2.

Table 5.2: Amount of Framework-related Activities

	eBay	**Marsh's**	**PeerTrust**	**REGRET**
#inheritance	2	2	2	2
#invocations	3	4	4	6
#overriding	2	3	2	2
#configFiles	1	2	1	1
#compDeployed	3	3	2	2

As explained in Section 5.4, each model requires inheriting from two framework classes, the class that determines the type of the entity, and the class that implements the model or the engine. The number of method invocations go hand in hand with the complexity of the model. Thus, REGRET requires up to 6 method calls whereas eBay only requires 3. The number of methods that need to be overridden is similar in all the models, although it is higher in Marh's model because it needs to implement the threshold value. All models require at least one configuration file with the self-adaptation policy, whereas Marsh's requires another one for setting the initial subjective factors of the entities. The deployment changes slightly, as in the case of centralized models (trust models like Marsh's and centralized reputation models like eBay's), three components must be deployed, whereas in distributed reputation models only two are

required[12].

As a conclusion, the framework entails negligible overhead (in the order of microseconds) and does not require a lot of work to implement well-known existing models. This means that the benefits of adopting the framework are quite high considering the work that the implementation requires (see Table 5.2) or even in terms of execution time, as shown in Figure 5.7. We advocate that these results make the adoption of the framework appealing.

5.6 Discussion

In this chapter we have developed a trust and reputation framework that allows implementing a wide range of trust and reputation models. The framework is implemented on top of a self-adaptive platform, which enables the use of trust and reputation information in order to make reconfiguration decisions. We have shown that the framework barely entails overhead and that the small amount of extra work for developers pays off given the interesting opportunities that the use of the framework brings.

Despite the huge amount of trust and reputation models proposed in the literature, we have found that by using only some core concepts (embodied in trust and reputation metamodels), it is possible to represent a wide range of them. This happens because the differences among models are often due to the application context where the models are proposed, rather than in their dynamics, which turn out to be similar in most cases.

We have learned that this kind of integration must overcome several challenges. In our view, one primary challenge is building a robust identity management system in order to uniquely identify trust and reputation entities, and to allow access at any moment to these identities. In our current implementation, we build upon the reflection layer of Kevoree so as to provide such identities. However, it would be desirable to keep track of entities that disappear and re-appear again in dynamic environments, which is something we do not tackle at the moment.

Second, more research on declarative reconfiguration policies is required. Current models@run.time platforms lack a usable mechanism to specify reconfiguration policies.

[12]In practice, three components should be deployed in order to test the reputation engines of PeerTrust and REGRET due to their consideration for groups and credibility.

Kevoree offers *Kevscript* instructions, which become cumbersome for advanced reconfigurations. We have provided a basic format to represent these policies, but this format may fall short of expressiveness as the complexity of scenarios increases.

We also find that models@run.time platforms should provide a great deal of usable low-level services in order to monitor certain aspects of the system, like the consumed resources by each component, the latency of communications, or the response times, because this information might be key to building robust trust and reputation models. Factor producer entities could use these services to monitor different aspects of the entities and feed the trust or reputation engines.

Chapter 6

Conclusions

In this thesis, we have proposed a set of methodologies, guidelines and tools for the engineering of trust and reputation into software systems. We have empowered systems engineers and developers with capabilities to support the inclusion of trust in the different activities of the SDLC.

We advocate that one of the reasons why there has been very little attention to trust by the security and software engineering communities is that there is a lack of insight on this concept, especially when applied in the computing domain. In order to remedy this, we have shed light on trust by gathering concepts that are related to it and by relating these concepts in a conceptual model.

The conceptual model elicits three sets of concepts. The first set corresponds to concepts that are common to every trust model, and includes concepts such as context, trust factors, trust purpose or the roles played by entities. The second and third sets refer to the two classes of models that we identify, namely decision and evaluation models. The former refers to those models that control access to resources by means of credentials and policies. The outcome of the model is typically a binary decision: access granted or access denied. Evaluation models focus on examining more closely the trust relationships among entities, determining to which extent the factors influence these relationships and analyzing the sources of information for a more informed trust evaluation. The outcome of these models is a trust value that is tagged to the trust relationships.

In addition to providing insight on trust and trust models, the conceptual model yields a framework for comparing different classes of trust models under a common basis.

We use this framework for comparing a wide spectrum of different classes of trust and reputation models: Marsh's model (75), PolicyMaker (18), Jøsang's belief model (55), REGRET (114), TrustBuilder (136), eBay reputation model (107), Falcone et al. (33), Trust−$\mathcal{X}$ (15), PeerTrust (137) and Agudo et al. (1).

The concepts gathered in the model serves us as a basis for the rest of the contributions of this thesis, as these concepts are used in the methodologies that we have proposed and the tools we have built for each phase of the SDLC.

As for the planning phase, we have proposed a methodology for evaluating trust in cloud providers, which is fundamental to making informed cloud sourcing decisions. The methodology allows the explicit representation of uncertainty and subjectivity, which were highlighted as key concepts related to trust, by means of confidence intervals and an operator for the aggregation of trust intervals that maintains the uncertainty along operations. Other important concepts identified in the conceptual model, such as sources of information and trust factors, are intensively used in order to gather knowledge about cloud providers and to evaluate trust in them. The outcome of the methodology consists of nine confidence intervals: one in the stakeholders dimensions (which measures trust in the staff and other stakeholders of the cloud provider); one in a general dimension (which measures general factors like transparency of the provider); and seven in the threats dimensions, one for each threat identified by the CSA. In order to validate the methodology, we have evaluated four popular cloud vendors: Amazon, Apple, Microsoft and Google. The context of the cloud sourcing is the management of EHRs, which contain private information about patients created by health care professionals. The results of applying the methodology have shown that no cloud vendor meet our trust expectations, although Microsoft was closer than the rest, followed by Apple, Amazon and Google.

The next phase of the SDLC, security analysis, comprises two main activities: security requirements elicitation and threats analysis. For the first activity, we have proposed a methodology and notation in order to capture and represent trust and reputation requirements and integrate these requirements with other functional and non-functional ones. The notation is an extension over problem frames, which focuses on representing the system in its context. In particular, we have extended UML4PF with many of the trust and reputation notions highlighted in the conceptual model. As part of the methodology, we have included support for OCL queries, which allows detecting

syntactic and semantic inconsistencies in the models. For example, every trust relationship must have a trustor and a trustee. If the engineer misses one of these roles, an OCL expression can detect and help fix this. In order to validate the methodology, we have used the protection profile defined by the Common Criteria for the home gateway in a smart grid setting. Discussions with security professionals have shown that our methodology is useful and can help in engineering trust-based security solutions.

As for the threat analysis, we have proposed a methodology and a tool for the identification of insider threats in socio-technical systems. The methodology includes a trust model, an asset model and a threat model. The former consists of a propagation model which, assuming that we have some pre-established trust relationships in the system, allows deriving the rest of trust relationships among entities. The model includes support for translating the numeric values into trust labels that are more easily interpreted by the security expert. The semantics of the trust relationships refer to the trust of permission over a resource owned by the trustor. Therefore when entity A trusts another entity B with regards to resource R, it means that A believes that B will not misuse his permissions in order to violate some of the security properties held by R. The asset model measures the criticality of resources by assigning them a sensitivity value, whereas the threat model ties both the trust and asset models together. Concretely, a threat is deduced to occur if an entity is granted a permission on an asset that is sufficient to violate the security property associated to the asset, and if this entity is not trusted by the owner of the asset with that permission. The severity of the threat is calculated taking the sensitivity level of the asset into account. The methodology has been validated in the context of an eHealth system that spans across hospitals and pharmacies, and which includes several roles including patients, physicians, and pharmacists.

For the secure design phase, we have proposed an extension over UML that describes trust and reputation models and its integration in the system. This UML extension builds upon many of the concepts identified in the conceptual model in order to extend three diagrams. Use case diagrams are intended to represent trust relationships and basic reputation information at a glimpse. We have also embedded information that allows making the purpose of trust more explicit at the use case level. Class diagrams are extended in order to provide more insight on the model, including the way trust and reputation are updated, the factors that are used for such update, and the way they

are to be represented in the system. The extensions over deployment diagrams allow representing trust at the infrastructure level and to determine where certain trust-related information (e.g. reputation scores) are to be stored in the system. Even when we do not provide extensions of behavioural diagrams, we suggest the use of activity diagrams and their swim lanes representation in order to clarify the interaction patterns between the system and its trust and reputation models. We have validated this approach in an eHealth scenario of patient monitoring, in which patients wear devices capable of measuring their vital signs and of sending them to the hospital servers, thanks to which physicians can monitor their patients remotely. We have defined the trust relationships between patients and physicians, between the physicians and the wearable devices, and how patients can rate physicians to yield reputation scores for the latter.

Once we have provided enough details about the trust or reputation model, this information can be fed to the secure implementation phase, for which we have described a development framework that can be used in order to implement a wide variety of trust and reputation models. Again, this framework builds upon the notions discussed in the conceptual model. First, we have enumerated a set of requirements that a trust or reputation framework should support and then we have described both a high-level and a low-level architectures that support the requirements. After providing guidelines for implementing the different components of the architecture, we have validated the framework in a social cloud scenario. In this scenario, cloud providers participate in a market of web services in which each provider can produce new services or consume services from others. Trust relationships among providers as well as the providers' reputation must be updated according to the actions of the providers, who can rate the services they consume. Traditionally, developers would feel unarmed when faced with the development of such scenario, but our framework supports the smooth implementation and integration of these trust functionalities.

Once a system is deployed with built-in trust and reputation capabilities, we can exploit the trust information to drive the evolution of the system at runtime. This corresponds to the last phase of the SDLC, and we have accomplished this by building a trust and reputation framework on top of Kevoree, a self-adaptive platform that implements the models@run.time paradigm. This paradigm represents a synchronized model of the the running system, in such a way that any change in the model translates

in an automatic change of the system. Our trust and reputation framework provides developers with the mechanisms to implement trust-aware systems, where system components can establish trust relationships with others. These components can also hold reputation values and provide feedback after the interaction with other components in order to update the trust relationships or the reputation scores. The fact that this trust-related information is available within the models@run.time paradigm implies that it can be used for driving self-adaptation decisions. Therefore, if a given trust relationship between two system components, or the reputation value of a component, falls bellow a specified threshold, corrective actions can be executed, such as the runtime removal of the component or its substitution for another one. We provide a simple policy language where developers can specify this kind of actions and the conditions under which they are triggered. In order to validate the framework, we implement several trust and reputation models on top of a distributed chat application. The implemented models are eBay (108), Marsh's (75), PeerTrust (137) and REGRET (114). We prove that the framework entails negligible overhead for every implemented model, in the order of microseconds, and that it does not require a lot of work to implement well-known models. In particular, all it is needed in most cases is inheriting from a couple of classes, and using or overriding two or three methods.

We identify several research questions that remain open for further study. Some of them were mentioned in their corresponding chapters, but are summarized here for the reader's convenience.

Integrated model-driven methodology for trust engineering Model-driven engineering has brought lots of benefits to the software engineering community, smoothing the construction process through abstractions that make it easier for developers to reason about the software and minimizing the need to write boiler plate code. Along this thesis, we have dealt with models at different phases of the life cycle, covering design time and runtime models of the system. In order to ensure that the running trust and reputation models adhere to the specification of the initial models, we need to fill the gaps among phases, defining transformations that preserve the properties and the semantics of the models. This would simplify greatly the building process and at the same time it would ensure the correctness of the trust solutions by construction.

High-level reasoning in self-adaptation policies Current models@run.time platforms lack a usable mechanism to specify reconfiguration policies. Kevoree, the platform on top of which we implement the trust and reputation framework, provides *Kevscript* instructions, which become cumbersome for advanced reconfigurations.

We have provided a basic format to represent these policies, but this format may fall short of expressiveness as the complexity of scenarios increases. In particular, these policies refer to very concrete situations, which is not adequate for more complex and dynamic environments where conditions cannot be so easily anticipated. In particular, there is a need for more general policies that allow reasoning about the environment at higher level of abstractions. For example, instead of a rule that states the following: "if the reputation of a console component falls below 2, then substitute it for a secure console component", there should be a rule that states that: "if a console behaviour seems suspicious, then secure the consoles with which it is talking". The framework should be able to reason what it takes to consider the behaviour of a console strange, and to identify the mechanisms that it can apply in order to protect the rest of consoles from the suspicious one. Research on intrusion detection systems, anomaly detection and artificial intelligence can provide the required know-how for solving these issues.

Integration of trust and reputation requirements and design methodologies into well-established practices There are well-established methodologies and notations for eliciting security requirements and for building security artifacts in the context of the system. These include methodologies like Secure Tropos, the Microsoft Security Development Life cycle and notations like UMLSec or SecureUML. Instead of providing yet another methodology or notation, it would be more interesting to investigate how new practices for including trust and reputation could integrate into existing solutions. This would remove the burden of having to learn new methodologies and would minimize the friction of trying to reconcile the traditional system design with a trust-aware system design.

Extensions and integration of the UMLTrep profile The UML profile that we present in order to specify initial trust and reputation solutions focuses on evaluation models, and therefore it captures some of the most relevant trust and reputation concepts that are found in such models. However, decision models and propagation models

entail other notions that should be captured. In the first case, we should encompass the concepts of policies, credentials, compliance checker, or trusted third parties. For the latter, it should be possible to describe how trust is to be transferred among entities, defining the operators that are to be used along trust chains and in different trust paths.

Given the close relationship between trust and security, it would be optimal if our profile was integrated with another profile that captured the specification of security solutions, such as UMLSec. This would provide a better overall picture of the security and trust of the system, and would allow specifying explicitly the relationship among trust and security and how they affect each other.

Configuration and visual support for trust and reputation implementation There is a trend towards configuration-based implementation libraries and frameworks because it boosts the productivity by reducing the need to write boiler plate code and by allowing developers to focus on the core functionality of the software. Visual tools also can be quite effective at improving the developers' productivity as it minimizes the need for writing code, which is always subject to compilation and runtime errors.

Although we struggle to provide fast to learn and easy to use trust and reputation frameworks, they are still code intensive. Therefore, we consider that one of the first improvements should go in the aforementioned direction. This would be especially effective if the frameworks were integrated in a higher-order model-driven workflow that allowed the automatic derivation of implementation entities from design artifacts, as discussed in the first point.

Some degree of decoupling of expertise from requirements elicitation and design In general, requirements elicitation, threats and risks analysis frameworks largely depend on the level of expertise of the engineers. This means that two systems that are meant to have the same set of requirements and which will operate in the same context under the same conditions may end up having different levels of security just because different engineers worked on them.

One logical way to minimize this human dependency is by having more engineers revising others' contributions and by forcing the statement of the rationale for the presence of each requirement and for every design decision. However, it is easy to make implicit assumptions that do not end up documented and that are eventually lost.

Another way is by providing tool-supported structured methodologies that are simple and intuitive enough. However, we think that some level of expertise is required and cannot be fully removed.

Reasoning engines for supporting software and security engineers on eliciting trust requirements Some of the concerns raised when we presented our trust requirements elicitation methodology referred to the difficulty in and time required for reading and understanding the output of the OCL expressions. One way to mitigate this problem is by implementing reasoning engines that support the software and security engineers during the requirements stage. Upon execution of an OCL expression, the reasoning engine can read and interpret the output, and if any error occurs, mitigation actions can be recommended to the engineers. In order for the reasoning engine to be the most effective, it should be designed in such a way that it can be tailored to the domain of the software that it is to be built. Semantic technologies such as ontologies can be used for such purpose.

Repositories of public information about cloud vendors There is a difficulty in finding information about cloud vendors, probably due to marketing interests and to an initial lack of transparency in the cloud environment. However, in order to evaluate trust in cloud vendors, it is required gathering as much information as possible about them, about how they operate, about their staff and stakeholders, about their policies and, in general, about the general satisfaction of other customers. In the approach that we propose in this thesis for the evaluation of trust in cloud providers, we assume that much of the information can be elicited, but in many cases this proves to be difficult, especially when evaluating the stakeholders dimension. Cloud providers may be also reluctant to reveal the security incidents suffered by other customers, therefore it would be necessary to count on a public reputation repository where customers could rate their experience with the providers. This would raise other challenges, as the need for ensuring fair ratings and avoiding collusions from competitors.

There are other research questions that, even though not so closely related to our work, they still may provide relevant inputs as well as general insight on trust engineering. Robust identity management systems must be in place in dynamic environments

to ensure that entities cannot change their identities in order to delete their bad reputation and pervert the trust models. It would be also interesting to evaluate how each trust relationship at the inner levels (e.g. among two system components) impacts and contributes to the trust relationships at the highest level (i.e. between the end-users and the system). It remains an open problem to define metrics that evaluate to which extent a trust model can improve the security of the system. This is also a tough problem because measuring security is itself an open problem currently. Finally, it would be desirable to be capable of measuring to which extent the quality of experience of end-users improve as a consequence of engineering trust and reputation solutions into systems. This would provide companies with a well-founded justification for budget allocation in this task along the SDLC, provided that we could find a positive correlation between the final users satisfaction and trust engineering practices.

Appendix A

Resumen en español

En este capítulo se presenta un resumen de las contribuciones de la tesis en español. Las dos primeras secciones introducen el tema de investigación y esbozan las contribuciones en el marco de dicho tema. El resto de secciones resume cada una de las principales contribuciones.

A.1 Marco de la tesis y objetivos

El campo de estudio en el que se enmarca la tesis es el de ingeniería de sistemas seguros (SSE en sus siglas en inglés). Esta disciplina combina dos campos de estudio: la seguridad y la ingeniería del software. El objetivo es cambiar el enfoque tradicional del tratamiento de la seguridad, el cual se basa en aplicar medidas correctivas cuando se detectan problemas. Por el contrario, el enfoque de la SSE es proactivo, en el sentido de que se busca considerar e integrar medidas de seguridad a lo largo de todo el ciclo de vida de un sistema (SDLC, de sus siglas en inglés), el cual se representa en la figura A.1. La idea es minimizar la superficie de ataque del sistema y así minimizar el número de incidentes de seguridad que pueden ocasionarse una vez el sistema esté en uso.

Típicamente, en cada fase se realizan un conjunto de actividades. Durante la primera fase, planificación, se estudia la viabilidad del sistema y se toman algunas decisiones preliminares sobre su futuro desarrollo. En la fase de análisis de seguridad, se estudian posibles amenazas al sistema y se definen los objetivos de seguridad de los distintos grupos interesados en el mismo[1]. A continuación se definen artefactos de diseño y la

[1] A partir de ahora nos referiremos a estos grupos con su denominación inglesa: stakeholders

Figure A.1: Ciclo de vida del desarrollo de sistemas seguros

arquitectura que da soporte a los servicios de seguridad que previenen las amenazas y que cubren los requisitos identificados en la fase anterior. Durante la fase de implementación, se utilizan guías y buenas prácticas de programación segura y se utilizan plataformas de ejecución seguras. La última fase consiste en la monitorización del sistema en tiempo de ejecución para garantizar que se están cumpliendo los requisitos y políticas de seguridad. De manera transversal al ciclo y controlando el flujo de las distintas actividades, se realizan operaciones de gestión de riesgos y de garantías de seguridad. Las primeras identifican y cuantifican riesgos que pueden poner en peligro el desarrollo del sistema, mientras que las últimas se encargan de garantizar que los objetivos y requisitos de seguridad se mantengan a lo largo del ciclo.

En la presente tesis se discute sobre ingeniería de confianza en lugar de ingeniería de seguridad. Esto significa que en lugar de considerar seguridad en las distintas fases del ciclo de vida, vamos a considerar aspectos de confianza a través de modelos de confianza y reputación. El concepto de confianza es más amplio que el de seguridad, y la relación entre ambos puede resumirse en los siguientes puntos:

- La presencia de seguridad no implica la de confianza. Sin embargo, el hecho de que los usuarios de un sistema crean que el mismo es seguro, en la mayoría de los casos aumentará su confianza.

- La presencia de confianza no implica la de seguridad. Los usuarios pueden confiar

en el sistema debido a otras propiedades: buena usabilidad, experiencias pasadas satisfactorias, etc.

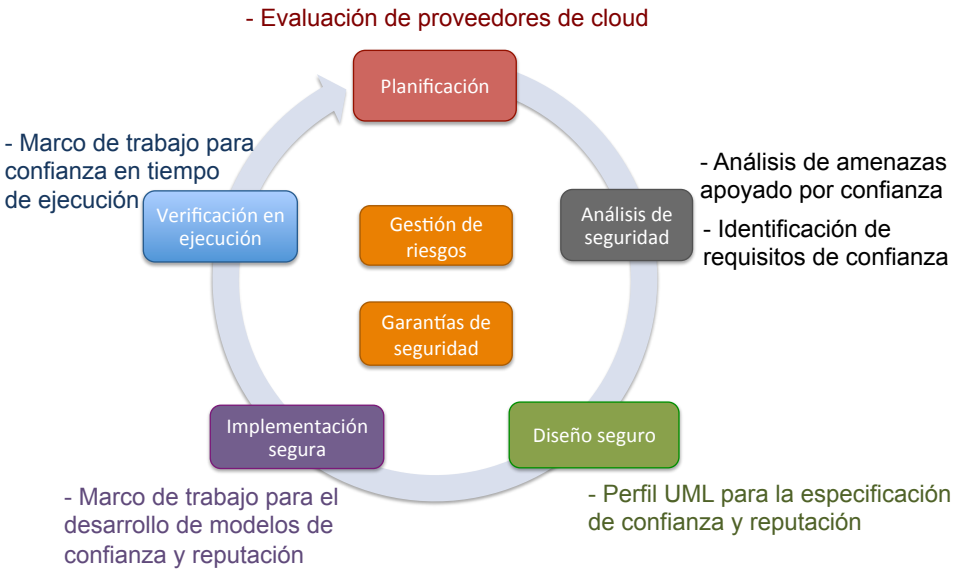
- La seguridad es una propiedad objetiva, mientras que la confianza es subjetiva. Dos sistemas pueden ser igual de seguros y aún así ser confiados de forma diferente por un mismo grupo de usuarios.

En el dominio computacional, la confianza se construye a partir de modelos de confianza y reputación. Así pues, el objetivo fundamental de la presente tesis es *el desarrollo de herramientas y metodologías que permitan el uso y la integración de modelos de confianza y reputación a lo largo del ciclo de vida de un sistema.*

A.2 Resumen de contribuciones

Dado que el objetivo es la integración de confianza en las distintas fases del ciclo de vida, gran parte de las contribuciones de la tesis están alineadas con dichas fases, como se observa en la figura A.2

Figure A.2: Contribuciones



Más concretamente, las contribuciones se listan a continuación:

- Estudio exhaustivo de la literatura existente en la integración de confianza en las distintas fases del SDLC.

- Análisis sistemático de múltiples definiciones de confianza para remarcar los componentes que las constituyen.
- Marco de trabajo conceptual que recoge conceptos relacionados con modelos de confianza y la relación entre dichos conceptos. Esto a su vez proporciona una base conceptual para comparar un amplio abanico de modelos de confianza y reputación.
- Metodología para incorporar un razonamiento guiado por la confianza durante la fase de planificación en el contexto de la evaluación de proveedores de cloud.
- Soporte metodológico y de herramientas para la identificación y análisis de amenazas en sistemas y organizaciones basados en el análisis de relaciones de confianza.
- Metodología y notación para la recogida de requisitos de confianza y reputación y para su integración con otros requisitos funcionales y no funcionales, incluidos los de seguridad.
- Notación que permite la especificación de modelos de confianza y reputación en el sistema.
- Marco de trabajo que permite la implementación de un amplio abanico de modelos de confianza y reputación.
- Marco de trabajo que permite construir sistemas que evolucionan en tiempo de ejecución de acuerdo a los valores de confianza y reputación de sus componentes.

Cada contribución se describe con más detalles en las siguientes secciones.

A.3 Marco de trabajo conceptual de confianza

Previo a la integración de confianza y reputación en las distintas fases del SDLC, es preciso sistematizar el conocimiento de estos conceptos. Para ello, primero analizamos un conjunto de definiciones de confianza que distintos autores han dado en los últimos años. A partir de las múltiples definiciones de confianza, construimos un mapa de conceptos utilizando Wordle[2] (véase la figura 2.1). El análisis revela que el concepto más

[2] http://www.wordle.net/

importante es el de entidad, lo cual es obvio dado que la confianza no tiene sentido (al menos de forma práctica) si no hay entidades que confían ni en las que se confían. El contexto es otro concepto clave dado que la confianza depende mucho de éste. Otros conceptos importantes connotan cierta incertidumbre, como subjetivo, creencia, disposición o expectación, mostrando que la confianza implica incertidumbre sobre el comportamiento de una entidad. Es important remarcar que aunque el concepto de riesgo no está presente de forma explícita en las definiciones, una lectura atenta revela que está de forma implícita en casi todas ellas. Por ejemplo, McKnight (80) afirma que "... consecuencias negativas son posibles", mientras que Mayer (78) estipula que la confianza implica disposición de vulnerabilidad.

A modo de resumen, el concepto de confianza está presente cuando hay incertidumbre y riesgo en la interacción entre dos o más entidades que necesitan colaborar en un contexto determinado. Si la entidad que deposita confianza conoce de antemano el resultado de la interacción sin ninguna duda, o si dicho resultado no supone riesgo alguno para la entidad, la confianza no es precisa. Dado que no hay ninguna definición que cubra todos los conceptos que creemos más importantes, nuestra definición de confianza es la siguiente: *la confianza es la expectación personal, única y temporal que una entidad deposita en otra en cuanto al resultado de una interacción entre ellos que afecta a la primera entidad.*

Una vez analizado el concepto de confianza, revisamos un conjunto de artículos de estudio de distintos modelos de confianza, así como modelos de confianza y reputación ampliamente conocidos. De esta forma podemos identificar conceptos que son transversales a todos ellos y que nos permiten abstraernos de sus particularidades, lo que nos lleva a un marco de trabajo conceptual que permite comparar distintos modelos de confianza y reputación. A continuación exponemos un resumen de los conceptos más importantes y sus relaciones, el cual queda reflejado en las figuras 2.2, 2.3 y 2.4.

La confianza la calculan los modelos de confianza, los cuales han de tener al menos dos entidades que han de interactuar. Las entidades juegan un rol, o varios roles. En los casos más generales, estos roles son trustor (la entidad que deposita la confianza), y trustee (la entidad sobre la que la confianza es depositada). Otros roles posibles son testigos, que son entidades que dan su opinión sobre otras entidades en función de observaciones o experiencias personales. Una vez que tenemos un trustor y un trustee, decimos que hay una relación de confianza.

A. RESUMEN EN ESPAÑOL

Establecer una relación de confianza persigue un propósito, como el acceso, la provisión o la identificación de entidades. Los factores que afectan a la confianza son el contexto, las propiedades subjetivas del trustor y del trustee, como la honestidad, creencias o los sentimientos, y las propiedades objetivas del trustor y del trustee, como el comportamiento o la seguridad.

Un modelo de confianza puede asumir determinados comportamientos, como que las entidades sólo emitirán evaluaciones justas de otras entidades, o que existe una serie de valores iniciales. Un modelo también puede seguir distintos enfoques de modelado, como matemáticos, lingüísticos y gráficos.

Podemos distinguir dos clases de modelos de confianza: los modelos de decisión y los de evaluación. Los modelos de decisión usan políticas, que especifican condiciones bajo las cuales se concede acceso a un recurso. Las políticas están escritas en un lenguaje de políticas, el cual puede considerar la resolución de conflictos entre políticas. Las condiciones de acceso se expresan en función de credenciales, las cuales son afirmaciones sobre alguna característica de una entidad (su identidad, si es miembro de un grupo, etc). Las credenciales pueden tener distintos formatos, como certificados X.509 o XML. El chequeador de conformidad es el componente que une las credenciales y las políticas al encargarse de verificar qué credenciales satisfacen qué políticas. Algunos modelos de decisión también permiten la búsqueda de credenciales a través de cadenas de credenciales, así como la verificación de su validez.

Los modelos de negociación son una especialización de los modelos de decisión que añaden una estrategia de negociación para permitir que las entidades revelen sus políticas y credenciales poco a poco, hasta llegar a un punto de confianza válida. Algunos usan tipos de evidencias, que representan información sobre el proceso de negociación y que permiten la optimización de éste.

El otro tipo de modelos, los modelos de evaluación, suelen seguir un ciclo de vida de dos etapas. Primero, una fase de inicio es necesaria para asignar valores iniciales a las entidades del sistema. La tendencia de confianza se refiere a la propensión del modelo hacia valores más altos o más bajos en esta primera fase. La segunda fase se corresponde a un proceso de evaluación que asigna valores de confianza de acuerdo a determinados factores. Este proceso requiere la monitorización para proporcionar datos precisos sobre estos factores.

Las relaciones de confianza se etiquetan con valores de confianza que indican cuánto un trustor confía en un trustee. Este valor tiene una dimensión, que indica si es un único valor o es una tupla de valores. Los valores también tienen una semántica asociada, que viene determinada por dos dimensiones: objetividad y ámbito. La primera se refiere a si la medida de confianza proviene de un juicio subjetivo de una entidad o de la evaluación de la entidad siguiendo un criterio formal. La segunda hace referencia al número de factores que se tienen en cuenta para la medida de confianza.

En muchos casos, el modelo también incluye el proceso para definir un umbral de confianza, incorporando la decisión de confianza en el propio modelo. Si el valor es mayor que el umbral, se asume que el trustor confía en el trustee y que la interacción puede proseguir.

En los modelos de evaluación, el proceso de cálculo de confianza, y todos los conceptos asociados a éste son los más importantes, porque en cierta medida se convierten en una firma del modelo que la hacen diferente del resto. El proceso de evaluación utiliza métricas de confianza, las cuales a su vez usan factores como el riesgo o la experiencia pasada y los combinan para dar lugar a un valor final. Las métricas de confianza usan motores de computación, que determinan la forma en que los factores se combinan, y que van desde sumas y medias, hasta motores de lógica difusa o bayesiana.

Las fuentes de información que proporcionan valores para los factores incluyen la experiencia directa con la entidad, factores sociológicos y psicológicos y opiniones de terceras partes. Los modelos de reputación utilizan información pública de confianza de otras entidades para dar lugar a un valor de reputación. Estos modelos pueden ser centralizados, en los que una entidad se encarga de recoger y distribuir información de reputación, o distribuidos, en los que cada entidad es responsable de mantener un registro de valores de confianza en otras entidades, y mandar esta información al resto de entidades. Los modelos de evaluación suelen tener en cuenta la credibilidad de la información y su frescura, esto es, cómo de reciente es dicha información.

Los modelos de propagación son una subclase de los modelos de evaluación y asumen que existen un conjunto de relaciones de confianza. Su objetivo es calcular nuevas relaciones de confianza entre entidades que no han tendio una experiencia directa. Para ello, algunos modelos asumen que la confianza es transitiva y explotan esta propiedad. Los nuevos valores de confianza se calculan mediante dos operadores: un concatenador

y un agregador. El primero calcula la confianza a través de una cadena, mientras que el segundo agrega los valores de todas las cadenas que llegan hasta la entidad objetivo.

Los modelos de reputación tienen su propia terminología, y basamos dicha terminología en los sistemas de reputación web. El concepto central es el de declaración de reputación, que es una tupla con un origen, una afirmación y un objetivo. Un origen es una entidad en el sistema que puede emitir afirmaciones sobre otra entidad del mismo, a la que se denomina objetivo. Los modelos de reputación utilizan motores de reputación que toman como entrada declaraciones de reputación sobre un objetivo y que producen un valor de reputación para dicho objetivo.

Finalmente, todos los conceptos discutidos hasta el momento los podemos utilizar para catalogar y comparar distintos modelos de confianza y reputación, como puede verse en las tablas de la sección 2.2.4.

A.4 Evaluación de confianza de proveedores de cloud

Una consideración muy importante que hay que hacer cuando se diseñan sistemas ICT hoy en día es si el sistema, o una parte de él, debe moverse al Cloud, actividad que se conoce como subcontratación cloud (del inglés, cloud sourcing). Esta actividad conlleva una pérdida de control del sistema y, por consiguiente, aumentan las preocupaciones sobre la seguridad así como las amenazas a las que está expuesto. En este contexto, la primera decisión importante es qué proveedor de cloud utilizar. Para tal propósito, proponemos una metodología que pueda ayudar a los responsables a cuantificar su confianza en distintos proveedores y elegir aquél que mejor cumpla sus expectativas.

Nuestra metodología se construye sobre el conocimiento de trabajos previos y busca abordar las necesidades que actualmente no están cubiertas. En particular, aunque la confianza ya se ha incorporado en la evaluación de clouds, en la mayoría de los casos, el propósito de esta evaluación es la selección de servicios y no la de proveedores. Por otro lado, la mayoría de las contribuciones se centran en métricas en lugar de una metodología concreta. Por último, la incertidumbre y la subjetividad, que son intrínsecas al concepto de confianza, no suelen tenerse en cuenta.

El primer paso consiste en obtener información sobre el proveedor. Luego, se recogen y se cuantifican factores de confianza sobre los stakeholders del proveedor y sobre el

proveedor como un todo. En paralelo, especificamos umbrales de confianza que dependen de los requisitos del escenario. Estos umbrales son los valores mínimos de confianza que se esperan obtener para un escenario concreto. En el siguiente paso, los factores se agregan en tres dimensiones diferentes: una dimensión de stakeholders, una dimensión de amenazas y una dimensión general. Dicha agregación se realiza mediante un operador de suma que definimos. Finalmente, la información se representa gráficamente.

Para cada dimensión, se utilizan unas plantillas en las que hay que evaluar distintos factores. Por ejemplo, en la dimensión de stakeholders, se puede comprobar cuántos años de experiencia lleva acumulados cada stakeholder, mientras que para la dimensión general, que se refiere al proveedor en su conjunto, se puede evaluar la transparencia de éste comprobando su página web y sus informes de incidentes, siempre y cuando esto sea posible. En la dimensión de amenazas, consideramos las amenazas enumeradas por el CSA (véase Tabla 3.3).

A la hora de cuantificar estos factores, utilizamos dos valores: el valor del factor, y un valor de credibilidad. Este último hace referencia a cómo de seguros estamos de que el valor del factor es preciso. De esta forma, estamos haciendo explícita la incertidumbre que se genera como consecuencia de tener información parcial y subjetiva. En particular, para ambos valores usamos números enteros entre 0 y 3, a partir de los cuáles formamos un intervalo de confianza $TI = [\frac{vc}{3}, \frac{vc}{3} + (3 - c)]$ en donde v es el valor del factor y c el valor de credibilidad.

Para reducir el número de intervalos y así poder procesar mejor la información, definimos un operador de suma de intervalos de confianza (ver Definición 3), según el cual el intervalo resultante de una suma está a medio camino entre los dos sumandos. Asimismo, el intervalo $[0, 3]$ representa la identidad del operador, lo cual es lógico dado que este intervalo representa la máxima incertidumbre y no añade nuevo conocimiento.

La metodología la hemos aplicado para la evaluación de los siguientes proveedores: Google, Amazon, Apple y Microsoft. El contexto de aplicación es el de un sistema de eHealth, en particular de EHRs, los cuales queremos mover al proveedor. Los resultados después de nuestro análisis muestran que no hay ningún proveedor que cumpla nuestras expectativas, aunque Microsoft sale mejor parado que sus competidores, especialmente en la dimensión de amenazas y en la dimensión general.

A.5 Uso de confianza durante la fase de análisis de seguridad

En la fase de análisis de seguridad se realizan fundamentalmente dos actividades: identificación de amenazas y análisis de requisitos. En cuanto a la primera, muchas amenazas, y especialmente aquéllas que resultan en los incidentes más graves, surgen a raíz de suposiciones falsas o implícitas sobre las relaciones de confianza en la organización donde el sistema se despliega. Por tanto, en esta tesis se propone una metodología que explícitamente muestra y analiza estas relaciones de confianza, a partir de las cuales es posible detectar posibles amenazas a distintas propiedades de seguridad de los recursos del sistema, como confidencialidad, integridad y disponibilidad (Confidentiality, Integrity, and Availability (CIA) de sus siglas en inglés).

Respecto a la captura de requisitos, mientras que la comunidad investigadora se ha centrado en desarrollar herramientas y notaciones para identificar los requisitos de seguridad tradicionales, poco se ha avanzado en la captura de requisitos de confianza y reputación. Para ello, proponemos una metodología y una extensión del enfoque de marcos de problemas (problem frames), el cual permite dar una representación de alto nivel del sistema en su contexto.

Las siguientes secciones profundizan más en cada una de estas contribuciones.

A.5.1 Identificación de amenazas internas guiada por relaciones de confianza

La metodología que proponemos permite a los ingenieros de seguridad identificar amenazas internas[3]. El enfoque consiste en primero modelar a los stakeholders del sistema, sus objetivos, los activos, las propiedades de seguridad (CIA) que los stakeholders quieren que se cumplan para sus activos, los permisos que los stakeholders tienen sobre sus activos, y las relaciones de confianza y los permisos entre ellos. Una relación de confianza de permisos repesenta la creencia que el otorgante tiene en que el receptor no dará mal uso a los permisos concedidos. El nivel de confianza asignado a un agente en relación a un permiso concedido es muy importante de cara a evaluar el riesgo de que dicho agente sea una amenaza interna: cuanto más baja sea la confianza, más probable

[3]Llamamos amenazas internas, del inglés *insider*, a las amenazas potenciales causadas por un trabajador de la organización que tiene información privilegiada sobre la misma.

es que el agente abuse de los permisos de acuerdo a la percepción del otorgante de los permisos.

Para apoyar la detección automática de amenazas internas, extendemos el lenguaje de modelado de requisitos SI* con un modelo de activos y un modelo de confianza. El modelo de activos asocia cada activo con un valor de *sensibilidad* que representa el valor del activo para el dueño. El modelo de confianza extiende el modelo de confianza binario de SI* (*confiar, no confiar*) y permite asociar diferentes niveles de confianza con un permiso concedido a un agente. En función de los niveles de sensibilidad y confianza, definimos un conjunto de reglas que permiten identificar automáticamente amenazas internas a un activo y priorizarlos basándonos en el riesgo asociado a cada amenaza. El riesgo asociado a una amenaza interna viene dado por la probabilidad de que la amenaza ocurra y por el coste de que se abuse del permiso.

En el modelo de confianza que proponemos, los niveles de confianza se pueden representar de dos formas: como números en el intervalo $[0, 1]$ y como etiquetas (por ejemplo, *Bien*, *Neutral* y *Mal*). Asumimos que algunos valores de confianza ya se han asignado a relaciones de confianza de permisos entre agentes en el modelo SI*. Estos valores son utilizados por el ingeniero de requisitos para calcular nuevos valores de confianza entre pares de agentes para los que no existe relación de confianza. En concreto, el nivel de confianza que el agente A deposita en otro agente B en relación a los permisos que le concede, se calcula a través de relaciones de confianza de permisos que otros agentes tienen con A y B. Así pues, de acuerdo a lo explicado en la sección A.3, lo que proponemos es la integración de un modelo de propagación en SI*, y por lo tanto, definimos un operador de concatenación y un operador de agregación. Una vez obtenemos un valor final para cada relación de confianza de permisos, usamos reglas de transformación para traducir estos valores desde el intervalo $[0, 1]$ a una etiqueta en una escala de confianza.

El modelo de amenaza que definimos considera que un agente A es una amenaza interna para un recurso S cuando se cumplen las siguientes condiciones:

- A A se le concede un permiso PT sobre el recurso S que es suficiente para violar alguna de sus propiedades de seguridad.
- El agente que es dueño del recurso S no confía en A con el permiso PT.

La gravedad de la amenaza depende de los niveles de sensibilidad de los activos y de los niveles de confianza de las relaciones de confianza de permisos.

Todo el proceso se apoya sobre una herramienta de SI* que está desarrollada como una extensión de Eclipse y que consta del motor de inferencia DLV. El interfaz de la herramienta permite dibujar un modelo SI* que es traducido automáticamente en una especifación ASP. La herramienta permite introducir las reglas para la identificación de amenazas internas.

A.5.2 Recogida de requisitos de confianza y reputación

Los marcos de problemas son un enfoque y notación para describir problemas de desarrollo del software. En los marcos de problemas, una máquina representa el software a desarrollar, mientras que un dominio es parte del mundo donde la máquina se instalará. Hay distintos tipos de dominios que representan a personas, datos y leyes físicas. La tarea de un ingeniero es construir la máquina descrita a través del enfoque del marco de problemas que mejore el comportamiento del entorno en el que se integra, de acuerdo a unos requisitos. Los marcos de problemas ayudan al ingeniero a entender qué problemas hay que resolver, qué dominios deben considerarse y qué conocimiento debe describirse para analizar el problema en profundidad.

El desarrollo de software con marcos de problemas se desarrolla de la siguiente manera. Primero, el entorno en el que la máquina operará se representa mediante un diagrama de contexto, que consiste en dominios e interfaces, pero que no incluye requisitos. Los diagramas de conocimiento de dominio se centran en un dominio en particular del diagrama de contexto e identifican más conocimiento sobre el dominio en términos de hechos y suposiciones. A continuación, el problema se descompone en subproblemas, cada uno de los cuales se representa con un diagrama de problema que contiene dominios, fenómenos, interfaces y sus relaciones con al menos un requisito.

UML4PF[4] es un perfil UML que extiende los diagramas de clase con estereotipos que representan los conceptos de los marcos de problemas. Nuestro trabajo consiste en extender UML4PF con nociones de confianza y reputación (véase la Figura 3.6), así como proponer una metodología que permita identificar cómo los requisitos de confianza y reputación encajan en el sistema y cómo se relacionan con sus requisitos (véase la

[4] `http://www.uml4pf.org`.

figura 3.7). Una ventaja de usar UML es que permite a los ingenieros de requisitos consultar los modelos mediante OCL, lo cual puede resultar útil para detectar información errónea o simplemente falta de información.

La metodología consiste en una serie de actividades secuenciales. En primer lugar, el ingeniero del software y el experto del domino describen el contexto del desarrollo del software mediante un diagrama de contexto, el cual describe la máquina en su entorno usando dominios e interfaces entre ellos. Un conjunto de requisitos funcionales textuales se refiere a los dominios, y el ingeniero de seguridad identifica activos y requisitos de seguridad para ellos. A continuación, el experto del dominio y el experto en confianza tienen que trabajar juntos. El primero identifica diagramas de dominio sin contar con confianza, mientras que el segundo proporciona un diagrama de confianza inicial, en el cual se esbozan los aspectos de más alto nivel de confianza y reputación. Estos aspectos incluyen las entidades de confianza, sus relaciones de confianza, las afirmaciones y los factores de confianza. En el siguiente paso, los ingenieros del software y de confianza refinan la información en el diagrama de confianza. Los diagramas finales contienen información detallada sobre las relaciones de confianza y los aspectos de reputación, como los roles que juegan las distintas entidades, más información sobre las afirmaciones, los valores de confianza y los factores objetivos y subjetivos. Además, en esta fase se analiza cómo los motores de confianza y reputación se integran en el sistema y su relación con los requisitos. Finalmente, el modelo se consulta mediante OCL para buscar problemas o inconsistencias, como por ejemplo la falta de la entidad con el rol trustee[5] en una relación de confianza.

La notación y la metodología la hemos aplicado para el perfil de protección que el Common Criteria[6] define para la puerta de enlace del medidor inteligente en un escenario de smart grid.

A.6 Especificación de modelos de confianza y reputación

De cara a que los ingenieros de requisitos comprendan mejor los requisitos de confianza y reputación y que los diseñadores puedan especificar soluciones basadas en ellos, definimos un perfil UML, el cual complementa la contribución anterior al extender diagramas

[5]El rol de las entidades que son confiadas por otras entidades.
[6]`https://www.commoncriteriaportal.org`

de casos de uso, diagramas de clase, y diagramas de despliegue con conceptos de confianza y reputación. Asimismo, mediante diagramas de actividad, es posible representar cómo se comunica el sistema con los modelos de confianza y reputación definidos.

El objetivo de la extensión realizada sobre los diagramas de casos de uso es señalar de un vistazo las relaciones de confianza que existen entre las distintas entidades del sistema, así como las entidades que pueden emitir afirmaciones sobre otras, incluyendo de esta forma información de reputación. También se definen explícitamente el contexto de la confianza y qué casos de uso se ven afectados por la misma. Esto último permite reflexionar sobre las decisiones en las que influyen la confianza y la reputación.

Las extensiones sobre los diagramas de clase permiten profundizar en distintos aspectos de confianza y reputación, especialmente en cuanto a los motores de confianza y de reputación y los factores que dichos motores utilizan. Los factores pueden ser objetivos o subjetivos, se puede especificar la dimensión de un valor de confianza o reputación (es decir, el número de componentes de los que consta), y si el valor proviene de un proceso de monitorización o es directamente asignado por alguna entidad.

Los diagramas de despliegue se extienden para poder representar información de confianza y reputación a nivel de infraestructura. Por un lado, las plataformas y las redes pueden establecer relaciones de confianza entre ellas e incluso pueden tener un valor de reputación, lo cual es especialmente útil en entornos abiertos y distribuidos, donde un nodo de procesamiento (por ejemplo, un móvil o un servidor) puede elegir entre distintos nodos para pasarle cierta información. En los diagramas de despliegue también es importante especificar qué componente del sistema se encargará de almacenar los valores de reputación en modelos centralizados.

Como se ha mencionado anteriormente, los diagramas de actividad permiten especificar cómo interactúa el sistema con los modelos de confianza y reputación. Esto hace posible que los diseñadores se centren en los patrones de interacción y por tanto tengan que tomar decisiones sobre estos mecanismos de comunicación. En particular, consideramos que una representación basada en carriles[7] permite definir bien la responsabilidad de cada parte del sistema.

Todo lo anterior lo aplicamos a un escenario de eHealth en el que el objetivo es poder recoger datos de los pacientes independientemente de su localización. De esta forma, los pacientes pueden recibir opiniones de inmediato ante una situación crítica y pueden ser

[7]Del inglés *swim lanes*.

atendidos por los médicos en cualquier momento y lugar. Para ello, los pacientes llevan puesto algún dispositivo que mide sus constantes vitales y que envía esta información a los servidores de los hospitales. En este escenario, es importante definir las relaciones de confianza entre pacientes y médicos, entre los médicos y los dispositivos de medición, y es posible evaluar la reputación de los médicos en función del trato a los pacientes. Estos valores de confianza y reputación permiten iniciar procesos para el cambio de médico o el cambio de dispositivo, entre otros.

A.7 Marco de trabajo para la implementación de modelos de confianza y reputación

A pesar de que se han propuesto numerosos modelos de confianza y reputación, se han destinado pocos esfuerzos a ofrecer herramientas que faciliten a los desarrolladores la implementación de estos modelos. En particular, los modelos propuestos suelen presentarse bajo unas suposiciones y contextos muy concretos, por lo que no es fácil adaptarlos a entornos más generales.

Para solucionar este problema, proponemos un marco de trabajo que funciona como un servidor que media entre una aplicación cliente y un sistema de bases de datos, de forma que la aplicación solicita información de confianza y reputación de estas bases de datos, y permite actualizarlas con nuevos valores.

El marco de trabajo cumple los siguientes requisitos:

- Gestión de entidades: las entidades mantienen valores de confianza con otras entidades. El marco de trabajo debe asignar identificadores únicos a estas entidades y debe poder recuperar información de confianza y reputación de una entidad.

- Gestión de relaciones de confianza: las relaciones de confianza cambian a lo largo del tiempo. Nuevas relaciones de confianza pueden aparecer, mientras que otras pueden eliminarse, y los valores de confianza que se asocian a las relaciones van cambiando.

- Definición de motores de computación: los motores de computación se encargan de calcular valores de confianza y reputación de acuerdo al modelo. Es importante que los desarrolladores puedan definir sus propias métricas.

- Definición de eventos: los eventos que ocurren en la aplicación activan la comunicación con el marco de trabajo, por lo que es preciso que el desarrollador pueda definir eventos y la forma en la que el marco de trabajo se comporta cuando los recibe.

- Gestión de afirmaciones: el tipo y el valor de una afirmación determina el valor de reputación. Debe ser posible configurar afirmaciones que satisfagan las necesidades específicas de la aplicación.

- Gestión de factores: las métricas se componen de factores, por lo que de cara a definir métricas nuevas es preciso que los desarrolladores puedan definir nuevos factores.

- Tiempo e incertidumbre: estos factores juegan un papel muy importante a la hora de calcular la confianza y la reputación, por lo que el marco de trabajo debería ofrecer mecanismos a los desarrolladores para incluirlos como parte del proceso de computación.

- Separación de confianza y reputación: el marco de trabajo debe permitir a los desarrolladores tratar la confianza y la reputación como conceptos diferentes, aunque dada su fuerte relación, debe permitir también que la una se valga de la otra.

A partir de los requisitos y tomando como base los conceptos desarrollados en la sección A.3, definimos una arquitectura de alto nivel y otra de bajo nivel, las cuales pueden verse en las figuras 4.2 y 4.3, respectivamente. La primera se organiza en capas lógicas, donde cada capa usa servicios proporcionados por la capa inferior. Las capas definidas son una capa de modelo, que captura la información básica de los modelos que el desarrollador puede implementar, una capa relacional, que se centra en el modelado de las entidades y relaciones de confianza, una capa de computación, que captura las métricas, y una capa de definición por el usuario, a partir de la cual se pueden definir nuevas métricas y factores para éstas.

La arquitectura de bajo nivel comprende los componentes y las estructuras de datos que dan lugar a una implementación inmediata del marco de trabajo. Incluyen conceptos de más bajo nivel, como eventos, afirmaciones de reputación y de confianza, así como elementos arquitecturales de comunicación como colas, que permiten ofrecer un sistema

asíncrono. De esta forma, cuando la aplicación cliente envía un evento, el marco de trabajo lo encola y permite que la aplicación cliente continúe, avisándolo cuando el resultado esté disponible.

Por último, ofrecemos guías de implementación del marco de trabajo. En concreto, proponemos su implementación mediante JavaEE[8]. Esta decisión ofrece dos ventajas: por un lado, facilita el desarrollo al ser Java un lenguaje ampliamente utilizado y conocido por desarrolladores, y en segundo lugar portabilidad, ya que el marco de trabajo puede ejecutarse sobre cualquier plataforma y sistema operativo.

El escenario utilizado para la validación del marco de trabajo es el de un desarrollador que tiene que implementar un sitio web social para proveedores de cloud. Éstos pueden registrarse en el sitio, y una vez registrados, pueden publicar servicios web en el sitio así como una descripción completa del mismo, por ejemplo mediante WSDL. Los proveedores pueden buscar un servicio web de acuerdo a sus necesidades, y usar el servicio para componer servicios más complejos. Cuando un proveedor consume un servicio de otro proveedor, el último puede cobrar al primero en función del tipo o complejidad del servicio. Así, el sitio actúa como un mercado de software entre proveedores.

De cara a ofrecer confianza en el sitio web y minimizar riesgos para los proveedores, se puede identificar una serie de requisitos de confianza y reputación. Para empezar, hay que señalar que hay dos tipos de entidades en este escenario: proveedores de cloud y servicios web. Ambos pueden tener valores de reputación derivados de opiniones de otros proveedores. Por ejemplo, si un proveedor usa un servicio y percibe que el servicio no se ejecuta como debería, podría valorar negativamente el servicio, lo que a su vez afectaría negativamente a la reputación del proveedor. Además de reputación, los proveedores pueden establecer relaciones de confianza entre ellos. Como ejemplos más concretos de posibles requisitos de confianza que el marco de trabajo permite implementar, consideramos los siguientes:

- Los proveedores pueden evaluar servicios web con una, dos o tres estrellas. Cuando un proveedor evalúa un servicio web, afecta a la reputación del servicio web y a la relación de confianza entre el evaluador y el creador del servicio web.

[8] `http://docs.oracle.com/javaee/7/index.html`

- Los proveedores pueden evaluarse entre sí usando una afirmación de tipo *Me gusta* y *No me gusta*. Cuando los proveedores evalúan otros proveedores, sólo afecta a su relación de confianza, pero no a la reputación.

- Leer completamente el perfil de un proveedor incrementa la creencia del lector en la aptitud de dicho proveedor.

- Los valores de reputación en el contexto *WebServiceForOffice* deben normalizarse al intervalo $[0, 1]$ antes de enviarse a la base de datos de confianza, y deben denormalizarse al intervalo original del modelo antes de enviarse a la aplicación.

A.8 Marco de trabajo para la implementación de sistemas autoadaptativos en función de valores de confianza y reputación

Las contribuciones anteriores tratan con el modelado y la implementación de sistemas que utilizan modelos de confianza y reputación, en tiempo de diseño. Sin embargo, dos cambios importantes están llegando al mundo de las ICT que precisan de una perspectiva que vaya más allá del diseño. Por un lado, la visión de orientación a servicios permite las mejoras de la funcionalidad al instante, por lo que las aplicaciones son más dinámicas y requieren estrategias de adaptación rápidas que cumplan con nuevos requisitos y que se adapten a entornos cambiantes. Por otro lado, las fronteras entre el mundo virtual y el físico están desapareciendo con la llegada del Internet de los Objetos, donde los sensores y actuadores se integran en objetos de la vida diaria y se conectan mediante redes capaces de producir una gran cantidad da datos. Todo esto hace más borrosa la frontera entre el tiempo de diseño y el de ejecución ya que se vuelve muy difícil para los desarrolladores predecir todas las posibles circunstancias que rodean a un sistema durante su ejecución.

Los modelos en tiempo de ejecución[9] constituyen un enfoque de construcción de software dirigido por modelos que permite la adaptación en tiempo de ejecución de sistemas distribuidos y heterogéneos. Permite trabajar con abstracciones para tratar con cambios imprevistos. Sin embargo, los marcos de trabajo que siguen este paradigma

[9] Models@run.time

ofrecen un soporte muy limitado para aspectos de seguridad, lo que entorpece su uso en escenarios reales.

Para superar este obstáculo, proponemos un marco de trabajo que permite construir sistemas que toman decisiones de reconfiguración en tiempo de ejecución basándose en relaciones de confianza y valores de reputación. Dicho marco de trabajo se integra en Kevoree[10], un modelo de componentes distribuido que implementa el paradigma de modelos en tiempo de ejecución, permitiendo así a los componentes del sistema incluir confianza en sus tomas de decisiones.

Kevoree ofrece su propio marco de trabajo que permite a los desarrolladores implementar sistemas autoadaptativos en Java. Para ello, Kevoree ofrece anotaciones que los desarrolladores pueden utilizar para crear nuevos componentes de Kevoree y nuevos parámetros para estos componentes. Los parámetros son atributos que pueden cambiarse fácilmente en tiempo de ejecución. El despliegue de sistemas Kevoree se puede realizar de dos maneras: a través de un editor visual, donde el desarrollador puede desplegar los nodos (es decir, el hardware) y los componentes que se ejecutan en éstos; y a través de *Kevscript*, un lenguaje de scripting que permite reflexionar sobre el sistema, tal como añadir o eliminar nodos y componentes, y cambiar el valor de parámetros. Este lenguaje de scripting también se utiliza para cambiar el sistema durante su ejecución.

El proceso que seguimos para el desarrollo de nuestro marco de trabajo de confianza y reputación es el siguiente. En primer lugar, definimos dos metamodelos, uno de confianza y otro de reputación, en EMF[11]. Estos metamodelos constituyen el esqueleto básico de los conceptos necesarios para expresar confianza y reputación en los componentes. A partir de estos metamodelos, se genera una API interna, invisible a los desarrolladores, a través de la cual gestionamos la información de los modelos de confianza y reputación. Para los desarrolladores, definimos dos APIs más fáciles de utilizar, una para el desarrollo de modelos de confianza y otra para el de modelos de reputación. Típicas tareas que los desarrolladores tienen que realizar para la implementación de modelos es la herencia de distintas clases y la implementación de métodos abstractos.

De cara a controlar la evolución del sistema en función de la confianza y la reputación, ofrecemos un lenguaje de políticas básico que permite establecer las condiciones

[10] `http://kevoree.org`
[11] http://www.eclipse.org/modeling/emf/

bajo las que el sistema ha de reconfigurarse. En el caso de querer usar la reputación, los políticas siguen el siguiente esquema:

TipoComponente Condición Acción Argumentos

el cual implica que si la *condición* para el tipo de componente *TipoComponente* se cumple, la *acción* con *argumentos* debe ejecutarse.

Por otro lado, para la confianza, se utiliza el siguiente esquema de políticas:

TipoComponente1 TipoComponente2 Condición/Umbral Acción Argumentos

el cual significa que si la confianza que *TipoComponente1* deposita en *TipoComponente2* cumple la *condición* o no llega al *umbral*, la *acción* con *argumentos* debe ejecutarse. Muchos modelos de confianza definen el umbral por encima del cual la confianza se garantiza, y por tanto, en lugar de utilizar una condición establecida por el desarrollador, la condición viene dada por el propio umbral del modelo.

En cuanto a las acciones, ofrecemos dos: *eliminar* y *sustituir*, lo que permite eliminar el componente que cumple la condición (o el umbral) o sustituirlo por otro que venga especificado en los argumentos de la política.

Para validar el marco de trabajo, utilizamos una aplicación de chat distribuido, donde el funcionamiento es similar para todos los modelos: una consola recibe un mensaje de otra consola e inspecciona sus contenidos. Dependiendo de estos contenidos (por ejemplo, si detecta una palabra malsonante), proporciona un estímulo para el modelo de confianza o reputación, el cual puede venir dado en forma de afirmaciones o cambios en factores. Demostramos que es posible implementar distintos tipo de modelos de forma fácil y que la sobrecarga que conlleva su uso en términos de esfuerzo y rendimiento es mínima, como muestran la figura 5.7 y la tabla 5.2.

References

[1] Isaac Agudo, Carmen Fernandez-Gago, and Javier Lopez. A Model for Trust Metrics Analysis. In *Proceedings of the 5th International Conference on Trust, Privacy and Security in Digital Business (TrustBus'08)*, volume 5185 of *LNCS*, pages 28–37. Springer, 2008. 40, 42, 184

[2] Isaac Agudo, Carmen Fernandez-Gago, and Javier Lopez. A Scale Based Trust Model for Multi-Context Environments. *Computers and Mathematics with Applications*, 60:209–216, July 2010. 71, 74

[3] Shakeel Ahmad, Bashir Ahmad ad Sheikh Muhammad Saqib, and Rashid Muhammad Khattak. Trust Model: Cloud's Provider and Cloud's User. *International Journal of Advanced Science and Technology*, 44:69–80, July 2012. 50

[4] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Publishing, 2nd edition, 2008. 2

[5] Donovan Artz and Yolanda Gil. A survey of trust in computer science and the Semantic Web. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5:58–71, 2007. 5, 20, 22, 35

[6] Yudis Asnar, Tong Li, Fabio Massacci, and Federica Paci. Computer Aided Threat Identification. In *Proceedings of the 2011 IEEE 13th Conference on Commerce and Enterprise Computing*, CEC '11, pages 145–152, Washington DC, USA, 2011. IEEE Computer Society. 26, 79

[7] Yudistira Asnar, Paolo Giorgini, and John Mylopoulos. Goal-driven risk assessment in requirements engineering. *Requirements Engineering*, 16(2):101–116, 2011. 66, 67, 68

[8] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January 2004. 8

[9] Azeem Sarwar and Muhammad Khan. A Review of Trust Aspects in Cloud Computing Security. *International Journal of Cloud Computing and Services Science (IJ-CLOSER)*, 2(2):116–122, 2013. 50

[10] Moritz Y. Becker and Peter Sewell. Cassandra: Flexible Trust Management, Applied to Electronic Health Records. *IEEE Computer Security Foundations Workshop*, 0:139, 2004. 23

[11] Kristian Beckers, Isabelle Côté, Stephan Faßbender, Maritta Heisel, and Stefan Hofbauer. A pattern-based method for establishing a cloud-specific information security management system. *Requirements Engineering*, 18(4):343–395, 2013. 53

[12] Kristian Beckers, Stephan Faßbender, and Maritta Heisel. A Meta-Model Approach to the Fundamentals for a Pattern Language for Context Elicitation. In *Proceedings of the 18th European Conference on Pattern Languages of Programs (Europlop)*. ACM, 2013. Accepted for Publication. 52, 53

[13] Kristian Beckers, Maritta Heisel, Francisco Moyano, and Carmen Fernandez-Gago. Engineering Trust- and Reputation-based Security Controls for Future Internet Systems. Technical report, University of Duisburg-Essen, 2014. 88

[14] Kristian Beckers, Jan-Christoph Küster, Stephan Faßbender, and Holger Schmidt. Pattern-Based Support for Context Establishment and Asset Identification of the ISO 27000 in the Field of Cloud Computing. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*, pages 327–333. IEEE Computer Society, 2011. 53

[15] Elisa Bertino, Elena Ferrari, and Anna Cinzia Squicciarini. Trust$-\mathcal{X}$: A Peer-to-Peer Framework for Trust Establishment. *IEEE Transactions on Knowledge and Data Engenieering*, 16(7):827–842, 2004. 24, 42, 184

[16] Gordon Blair, Nelly Bencomo, and Robert B. France. Models@ run.time. *Computer*, 42(10):22–27, 2009. 144

[17] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. KeyNote: Trust Management for Public-Key Infrastructures (Position Paper). In *Proceedings of the 6th International Workshop on Security Protocols*, pages 59–63, London, UK, 1999. Springer-Verlag. 22

[18] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized Trust Management. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, 1996. 3, 6, 22, 26, 35, 42, 184

[19] P.A. Bonatti, J.L. De Coi, Olmedilla D., and L. Sauro. A Rule-Based Trust Negotiation System. *IEEE Transactions on Knowledge and Data Engineering*, 22(11):1507–1520, 2010. 23

[20] S. Boon and J. Holmes. The dynamics of interpersonal trust: Resolving uncertainty in the face of risk. *Cooperation and prosocial behaviour*, pages 190–211, 1991. 33

[21] BSI. Protection Profile for the Gateway of a Smart Metering System (Gateway PP). Version 01.01.01(final draft), Bundesamt für Sicherheit in der Informationstechnik (BSI) - Federal Office for Information Security Germany, 2011. `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/SmartMeter/PP-SmartMeter.pdf?__blob=publicationFile`. 91

[22] Marian Bubak, Marek Kasztelnik, Maciej Malawski, Jan Meizner, Piotr Nowakowski, and Susheel Varma. Evaluation of Cloud Providers for VPH Applications. In *Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'13)*, pages 200–201. IEEE, May 2013. 49

[23] Vinny Cahill, Elizabeth Gray, Jean-Marc Seigneur, Christian D. Jensen, Yong Chen, Brian Shand, Nathan Dimmock, Andy Twigg, Jean Bacon, Colin English, Waleed Wagealla, Sotirios Terzis, Paddy Nixon, Giovanna di Marzo Serugendo, Ciaran Bryce, Marco Carbone, Karl Krukow, and Mogens Nielsen. Using Trust for Secure Collaboration in Uncertain Environments. *IEEE Pervasive Computing*, 2(3):52–61, July 2003. 28

[24] Christiano Castelfranchi and Rino Falcone. *Trust Theory: A Socio-Cognitive and Computational Model*. Wiley Publishing, 1st edition, 2010. 24

[25] Sudip Chakraborty and Indrajit Ray. TrustBAC: integrating trust relationships into the RBAC model for access control in open systems. In *Proceedings of the eleventh ACM symposium on Access control models and technologies*, SACMAT '06, pages 49–58. ACM, 2006. 27

[26] Sudip Chakraborty and Krishnendu Roy. An SLA-based Framework for Estimating Trustworthiness of a Cloud. In *Proceedings of the IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom'12)*, pages 937–942. IEEE, Jun 2012. 50

[27] Bruce Christianson and William S. Harbison. Why Isn't Trust Transitive? In *Proceedings of the International Workshop on Security Protocols*, pages 171–176. Springer-Verlag, 1997. 40

[28] Yang-Hua Chu, Joan Feigenbaum, Brian LaMacchia, Paul Resnick, and Martin Strauss. REFEREE: Trust Management for Web Applications. *Computer Networks and ISDN Systems*, 29:953–964, September 1997. 23, 26

[29] William Conner, Arun Iyengar, Thomas Mikalsen, Isabelle Rouvellou, and Klara Nahrstedt. A trust management framework for service-oriented environments. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 891–900, New York, USA, 2009. ACM. 30

[30] Isabelle Côté, Denis Hatebur, Maritta Heisel, and Holger Schmidt. UML4PF - A Tool for Problem-Oriented Requirements Analysis. In *Proceedings of the International Conference on Requirements Engineering (RE'11)*, pages 349–350. IEEE Computer Society, 2011. 85

[31] C. Crapanzano, F. Milazzo, A. De Paola, and G.L. Re. Reputation Management for Distributed Service-Oriented Architectures. In *Proceedings of the 4th IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshop (SASOW'10)*, pages 160–165, 2010. 30

[32] Golnaz Elahi, Eric Yu, and Nicola Zannone. A vulnerability-centric requirements engineering framework: analyzing security attacks, countermeasures, and requirements based on vulnerabilities. *Requirements Engineering*, 15(1):41–62, November 2009. 27

[33] Rino Falcone, Giovanni Pezzulo, and Cristiano Castelfranchi. A Fuzzy Approach to a Belief-based Trust Computation. In *Proceedings of the 2002 International Conference on Trust, Reputation, and Security: Theories and Practice*, volume 2631 of *LNCS*, pages 73–86. Springer-Verlag, 2003. 42, 184

[34] Randy Farmer and Bryce Glass. *Building Web Reputation Systems*. Yahoo! Press, USA, 1st edition, 2010. 9, 41, 140

[35] E. Fernández-Medina, J. Jurjens, J. Trujillo, and S. Jajodia. Editorial: Model-Driven Development for secure information systems. *Information and Software Technology*, 51(5):809–814, 2009. 2

[36] François Fouquet, Olivier Barais, Noël Plouzeau, Jean-Marc Jézéquel, Brice Morin, and Franck Fleurey. A Dynamic Component Model for Cyber Physical Systems. In *Proceedings of the 15th International ACM SIGSOFT Symposium on Component Based Software Engineering*, pages 135–144, Bertinoro, Italy, July 2012. 144

[37] Diego Gambetta. Can We Trust Trust? In *Trust: Making and Breaking Cooperative Relations*, pages 213–237. Basil Blackwell, 1988. 32, 33

[38] Jerry Gao, Pushkala Pattabhiraman, Xiaoying Bai, and W. T. Tsai. SaaS performance and scalability evaluation in clouds. In *Proceedings of the IEEE 6th International Symposium on Service-Oriented System Engineering (SOSE'11)*, volume 0, pages 61–71, Los Alamitos, CA, USA, Dec 2011. IEEE Computer Society. 49, 50

[39] Saurabh Kumar Garg, Steve Versteeg, and Rajkumar Buyya. SMICloud: A Framework for Comparing and Ranking Cloud Services. In *Proceedings of the 4th IEEE International Conference on Utility and Cloud Computing*, UCC '11, pages 210–218, Washington DC, USA, 2011. IEEE Computer Society. 51

[40] John C. Georgas, André van der Hoek, and Richard N. Taylor. Architectural Runtime Configuration Management in Support of Dependable Self-adaptive Software. *SIGSOFT Software Engineering Notes*, 30(4):1–6, May 2005. 31

[41] Carlo Ghezzi. The Fading Boundary between Development Time and Run Time. In *Proceedings of the Ninth IEEE European Conference on Web Services (ECOWS'11)*, page 11, Sep 2011. 143

[42] Anup K. Ghosh, Chuck Howell, and James A. Whittaker. Building Software Securely from the Ground Up. *IEEE Software*, 19(1):14–16, 2002. 2

[43] Tyrone Grandison and Morris Sloman. A survey of trust in internet applications. *IEEE Communications Surveys & Tutorials*, 3(4):2–16, 2000. 5, 18, 33, 36

[44] Tyrone Grandison and Morris Sloman. Trust Management Tools for Internet Applications. In *Proceedings of the 1st International Conference on Trust Management*, iTrust'03, pages 91–107, Berlin, Heidelberg, 2003. Springer-Verlag. 23, 26

[45] Nathan Griffiths. A Fuzzy Approach to Reasoning with Trust, Distrust and Insufficient Trust. In *Proceedings of the 10th International Conference on Cooperative Information Agents*, CIA'06, pages 360–374, Berlin, Heidelberg, 2006. Springer-Verlag. 49

[46] Sheikh Mahbub Habib, Vijay Varadharajan, and Max Mühlhäuser. A framework for evaluating trust of service providers in cloud marketplaces. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1963–1965. ACM, 2013. 51

[47] Haouas Hanen and Johann Bourcier. Dependability-Driven Runtime Management of Service Oriented Architectures. In *4th International Workshop on Principles of Engineering Service-Oriented Systems (PESOS'12)*, pages 15–21, Zurich, Switzerland, June 2012. 30

[48] Chern Har Yew. *Architecture Supporting Computational Trust Formation*. PhD thesis, University of Western Ontario, London, Ontario, 2011. 28, 33

[49] Denis Hatebur and Maritta Heisel. A UML Profile for Requirements Analysis of Dependable Software. In Erwin Schoitsch, editor, *Proceedings of the International Conference on Computer Safety, Reliability and Security (SAFECOMP'10)*, volume 6351 of *LNCS*, pages 317–331. Springer Berlin Heidelberg, 2010. 85

[50] Ferry Hendrikx, Kris Bubendorfer, and Ryan Chard. Reputation systems: A survey and taxonomy. *Journal of Parallel and Distributed Computing*, 75(0):184 – 197, 2015. 20

[51] P. Herrmann and H. Krumm. Trust-adapted enforcement of security policies in distributed component-structured applications. In *Proceedings of the 6th IEEE Symposium on Computers and Communications*, pages 2–8, 2001. 31

[52] Peter Herrmann. *Trust Management: First International Conference on Trust Management (iTrust'03)*, volume 2692 of *LNCS*, chapter Trust-Based Protection of Software Component Users and Designers, pages 75–90. Springer Berlin Heidelberg, May 2013. 31

[53] T.D. Huynh. A Personalized Framework for Trust Assessment. *ACM Symposioum on Applied Computing - Trust, Reputation, Evidence and other Collaboration Know-how Track*, 2:1302–1307, Dec 2008. 28

[54] Michael Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. 84

[55] Audun Jøsang. A logic for uncertain probabilities. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 9(3):279–311, June 2001. 40, 42, 184

[56] Audun Jøsang, Roslan Ismail, and Colin Boyd. A survey of trust and reputation systems for online service provision. *Decision Support Systems*, 43(2):618–644, March 2007. 20, 36, 39, 40, 63

[57] Audun Jøsang and Stéphane Lo Presti. Analysing the Relationship between Risk and Trust. In Christian Jensen, Stefan Poslad, and Theo Dimitrakos, editors, *Trust Management*, volume 2995 of *LNCS*, pages 135–145. Springer Berlin Heidelberg, 2004. 7, 8

[58] Jan Jürjens. UMLsec: Extending UML for Secure Systems Development. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, UML '02, pages 412–425. Springer-Verlag, 2002. 25

[59] Rolf Kiefhaber, Florian Siefert, Gerrit Anders, Theo Ungerer, and Wolfgang Reif. The Trust-Enabling Middleware: Introduction and Application. Technical report, Institut für Informatik Universität Augsburg, March 2011. 31

[60] L. Klejnowski, Y. Bernard, J. Hähner, and C. Müller-Schloer. An Architecture for Trust-Adaptive Agents. In *Proceedings of the 4th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO'10)*, pages 178–183. IEEE, 2010. 30

[61] R.K.L. Ko, P. Jagadpramana, M. Mowbray, S. Pearson, M. Kirchberg, Qianhui Liang, and Bu Sung Lee. TrustCloud: A Framework for Accountability and Trust in Cloud Computing. In *Proceedings of the 2011 IEEE World Congress on Services (SERVICES'11)*, pages 584–588, July 2011. 50

[62] Adam J. Lee, Marianne Winslett, and Kenneth J. Perano. TrustBuilder2: A Reconfigurable Framework for Trust Negotiation. In Elena Ferrari, Ninghui Li, Elisa Bertino, and Yücel Karabulut, editors, *Proceedings of the 3rd IFIP WG 11.11 International Conference on Trust Management*, volume 300 of *IFIP Conference Proceedings*, pages 176–195. Springer, 2009. 24, 28

[63] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, July 2006. 69

[64] Eddie Li, Liam O'Brien, He Jason Zhang, and Rainbow Cai. A Practical Methodology for Cloud Services Evaluation. In *Proceedings of the 3rd IEEE International Workshop on the Future of Software Engineering for and in Cloud*, pages 44 – 51, Santa Clara Marriott, USA, June 2013. 51

[65] Tong Li, Lin Liu, and Barrett R. Bryant. Service Security Analysis Based on i*: An Approach from the Attacker Viewpoint. In *Proceedings of Security, Trust, and Privacy for Software Applications (STPSA'10)*, pages 127–133, Seoul, 2010. 27

[66] Zheng Li, Liam OBrien, Rainbow Cai, and He Zhang. Towards a Taxonomy of Performance Evaluation of Commercial Cloud Services. *Proceedings of the IEEE 5th International Conference on Cloud Computing*, 0:344–351, 2012. 51

[67] Lin Liu, Eric Yu, and John Mylopoulos. Security and privacy requirements analysis within a social setting. *Proceedings of the 11th IEEE International Requirements Engineering Conference*, 3:151–161, 2003. 27

[68] Torsten Lodderstedt, David A. Basin, and Jürgen Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, UML '02, pages 426–441. Springer-Verlag, 2002. 25

[69] L. Luca, D. Pierpaolo, B. Riccardo, B. Stephen, and B. Andrew. Enabling Adaptation in Trust Computations. In *ComputationWorld '09: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, pages 701–706. IEEE, 2009. 31

[70] Mass Soldal Lund, Bjørnar Solhaug, and Ketil Stølen. *Model-Driven Risk Analysis - The CORAS Approach*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. 26, 66

[71] M.S. Lund, B. Solhaug, and Ketil Stolen. Evolution in Relation to Risk and Trust Management. *IEEE Computer*, 43(5):49–55, May 2010. 7, 8

[72] Supriya M., Venkataramana L.J., K. Sangeeta, and G.K. Patra. Estimating Trust Value for Cloud Service Providers using Fuzzy Logic. *International Journal of Computer Applications*, 48(19):28–34, Jun 2012. Published by Foundation of Computer Science, New York, USA. 51

[73] Mary Madden. Public Perceptions of Privacy and Security in the Post-Snowden Era. Technical report, Pew Research Center, November 2014. 7

[74] Paul Manuel. A trust model of cloud computing based on Quality of Service. *Annals of Operations Research*, pages 1–12, 2013. 50

[75] Stephen Marsh. *Formalising Trust as a Computational Concept*. PhD thesis, University of Stirling, April 1994. 3, 6, 24, 35, 39, 42, 168, 184, 187

[76] William Martorelli, Chris Andrews, and Sean-Paul Mauro. Cloud Computing's Impact on Outsourcing Contracts, January 2012. 48

[77] Fabio Massacci, John Mylopoulos, and Nicola Zannone. Security Requirements Engineering : The SI* Modeling Language and the Secure Tropos Methodology. In Zbigniew Ras and Li-Shiang Tsay, editors, *Advances in Intelligent Information Systems*, volume 265 of *Studies in Computational Intelligence*, pages 147–174. Springer Berlin - Heidelberg, 2010. 26, 66, 67

[78] Roger C. Mayer, James H. Davis, and F. David Schoorman. An Integrative Model of Organizational Trust. *Academy of Management Review*, 20(3):709–734, 1995. 8, 33, 34, 197

[79] John McDermott and Chris Fox. Using Abuse Case Models for Security Requirements Analysis. In *Proceedings of the 15th Annual Computer Security Applications Conference*, ACSAC '99, page 55. IEEE Computer Society, 1999. 25

[80] D. Harrison McKnight and Norman L. Chervany. The Meanings of Trust. Technical report, University of Minnesota, Management Information Systems Research Center, 1996. 33, 34, 197

[81] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing. Working Paper of the National Institute of Standards and Technology (NIST), 2009. 48

[82] Keith W. Miller, Jeffrey Voas, and Phil Laplante. In Trust We Trust. *Computer*, 43:85–87, 2010. 32

[83] Haralambos Mouratidis and Paolo Giorgini. Secure Tropos: a Security-Oriented Extension of the Tropos Methodology. *International Journal of Software Engineering and Knowledge Engineering*, 17(2):285–309, 2007. 26

[84] Haralambos Mouratidis, Michalis Pavlidis, and Shareeful Islam. Modelling Security Using Trust Based Concepts. *International Journal of Secure Software Engeneering*, 3(2):36–53, April 2012. 27

[85] L. Mui, M. Mohtashemi, and A. Halberstadt. A Computational Model of Trust and Reputation. In *Proceedings of the 35th Hawaii International Conference on System Science (HICSS)*, pages 280–287, 2002. 33

[86] Neovise Research Report. Use of Public, Private and Hybrid Cloud Computing, 2013. 48

[87] NESSoS Consortium. Deliverable 11: Initial version of two case studies, evaluating methodologies. `http://www.nessos-project.eu/media/deliverables/y2/NESSoS-D11.3.pdf`, October 2012. 105

[88] Ninghui Li and John C. Mitchell and William H. Winsborough. Design of a Role-Based Trust Management Framework. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 114–130. IEEE Computer Society Press, May 2002. 23

[89] Zeinab Noorian and Mihaela Ulieru. The State of the Art in Trust and Reputation Systems: A Framework for Comparison. *Journal of theoretical and applied electronic commerce research*, 5(2):97 – 117, August 2010. 20

[90] David Nuñez, Carmen Fernandez-Gago, Siani Pearson, and Massimo Felici. A Metamodel for Measuring Accountability Attributes in the Cloud. In *Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom'13)*, Bristol, UK, 2013. IEEE. 49, 50

[91] D. Olmedilla, O.F. Rana, B. Matthews, and W. Nejdl. Security and Trust Issues in Semantic Grids. In *Proceedings of the Dagsthul Seminar, Semantic Grid: The Convergence of Technologies*, volume 5271, 2005. 33

[92] Ontario. Standards of Sound Business and Financial Practices. Enterprise Risk Management: Application Guide. Technical report, Deposit Insurance Corporation of Ontario, 2011. 63

[93] K. Orkphol and Li Jianli. Multi-negotiation targets in Automated Trust Negotiation over TrustBuilder2 framework. In *Proceedings of the 8th International Conference on Computing Technology and Information Management*, ICCM'12, pages 101–105. IEEE, 2012. 24

[94] Wayne Pauley. Cloud Provider Transparency: An Empirical Evaluation. *IEEE Security & Privacy*, 8(6):32–39, 2010. 49, 50

[95] Michalis Pavlidis. Designing for trust. In *Proceedings of the CAiSE Doctoral Consortium 2011*, volume 731 of *CEUR-WS*, Jun 2011. 8, 65

[96] Michalis Pavlidis, Haralambos Mouratidis, Christos Kalloniatis, Shareeful Islam, and Stefanos Gritzalis. Trustworthy Selection of Cloud Providers Based on Security and Privacy Requirements: Justifying Trust Assumptions. In *Proceedings of the 10th International Conference on Trust, Privacy, and Security in Digital Business (TrustBus'13)*, volume 8058 of *LCNS*, pages 185–198. Springer Berlin Heidelberg, 2013. 50

[97] S. Phoomvuthisarn, Yan Liu, and Jun Han. An Architectural Approach to Composing Reputation-Based Trustworthy Services. In *Proceedings of the 21st Australian Software Engineering Conference (ASWEC'10)*, pages 117–126, 2010. 31

[98] Isaac Pinyol and Jordi Sabater-Mir. Computational trust and reputation models for open multi-agent systems: a review. *Artificial Intelligence Review*, 40(1):1–25, 2013. 19, 40

[99] Ponemon Institute Research Report. Security of Cloud Computing Users Study. Technical report, Ponemon Institute, sponsored by CA Technologies, March 2013. 48

[100] Harald Psaier, Lukasz Juszczyk, Florian Skopik, Daniel Schall, and Schahram Dustdar. Runtime Behavior Monitoring and Self-Adaptation in Service-Oriented Systems. In *Proceedings of the IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems (SASO'13)*, volume 0, pages 164–173, Los Alamitos, CA, USA, 2013. IEEE Computer Society. 31

[101] PwC. Managing cyber risks in an interconnected world. Technical report, PwC, Sep 2014. 6

[102] Lie Qu, Yan Wang, and Mehmet A. Orgun. Cloud Service Selection Based on the Aggregation of User Feedback and Quantitative Performance Assessment. In *Proceedings of the IEEE International Conference on Services Computing*, SCC '13, pages 152–159, Washington DC, USA, 2013. IEEE Computer Society. 51

[103] Massimiliano Rak and Giuseppe Aversano. Benchmarks in the Cloud: The mOSAIC Benchmarking Framework. In *Proceedings of the 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'12)*, pages 415–422, Los Alamitos, CA, USA, September 2012. IEEE Computer Society. 50

[104] SarvaPali D. Ramchurn, Dong Huynh, and Nicholas R. Jennings. Trust in multi-agent systems. *The Knowledge Engineering Review*, 19(01):1–25, April 2005. 18, 29, 33, 104

[105] Ahmad Rashidi and Naser Movahhedinia. A Model for User Trust in Cloud Computing. *International Journal on Cloud Computing: Services and Architecture(IJCCSA)*, 2(2), 2012. 50

[106] Lars Rasmusson and Sverker Jansson. Simulated social control for secure internet commerce. In *Proceedings of the 1996 workshop on new security paradigms*, NSPW '96, pages 18–25, New York, USA, 1996. ACM. 29

[107] Paul Resnick and Richard Zeckhauser. Trust Among Strangers in Internet Transactions: Empirical Analysis of eBay's Reputation System. In Michael R. Baye, editor, *The Economics of the Internet and E-Commerce*, volume 11 of *Advances in Applied Microeconomics*, pages 127–157. Elsevier Science, 2002. 35, 42, 184

[108] Paul Resnick, Richard Zeckhauser, John Swanson, and Kate Lockwood. The value of reputation on ebay: A controlled experiment. *Experimental Economics*, 9(2):79–101, Jun 2006. 165, 187

[109] Paul Robertson and Robert Laddaga. Adaptive Security and Trust. In *Proceedings of the 6th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO'12)*, pages 55–60. IEEE Computer Society, 2012. 29

[110] Paul Robertson, Robert Laddaga, and Mark H. Burstein. Trust and Adaptation in STRATUS. In *Proceedings of the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom'13)*, pages 1711–1716. IEEE, 2013. 29

[111] Nico Rödder, Rico Knapper, and Jochen Martin. Risk in modern IT service landscapes: Towards a dynamic model. In *Proceedings of the 5th IEEE International Conference on Service-Oriented Computing and Applications (SOCA'12)*, pages 1–4. IEEE, 2012. 51

[112] S. Ruohomaa, L. Kutvonen, and E. Koutrouli. Reputation Management Survey. In *Proceedings of the 2nd International Conference on Availability, Reliability and Security (ARES'07)*, pages 103–111, 2007. 19

[113] Sini Ruohomaa and Lea Kutvonen. Trust Management Survey. In *Proceedings of the 3rd international conference on Trust Management*, iTrust'05, pages 77–92. Springer-Verlag, 2005. 4, 19, 33

[114] Jordi Sabater and Carles Sierra. REGRET: reputation in gregarious societies. In *Proceedings of the 5th international conference on Autonomous agents*, AGENTS '01, pages 194–195. ACM, 2001. 42, 176, 184, 187

[115] Jordi Sabater and Carles Sierra. Review on Computational Trust and Reputation Models. *Artificial Intelligence Review*, 24:33–60, September 2005. 18

[116] Bruce Schneier. Attack Trees: Modeling Security Threats. *Dr. Dobb's Journal*, Dec 1999. 25

[117] Hassan Shakeri, Abbas G.Bafghi, and Hadi S.Yazdi. Computing Trust Resultant using Intervals. In IEEE, editor, *Proceedings of the 8th International ISC Conference on Information Security and Cryptology (ISCISC'11)*, pages 15–20, 2011. 54

[118] G. Silowash, D. Cappelli, A.P. Moore, R.F. Trzeciak, T.J. Shimeall, and L. Flynn. Common Sense Guide to Mitigating Insider Threats. Technical Report CMU/SEI-2012-TR-012, Software Engineering Institute, Carnegie Mellon, December 2012. 65

[119] Guttorm Sindre and Andreas L. Opdahl. Eliciting security requirements with misuse cases. *Requirements Engeneering*, 10(1):34–44, January 2005. 25

[120] Software Engineering Institute. 2011 CyberSecurity Watch Survey. Technical report, Software Engineering Institute, Carnegie Mellon, 2014. 65

[121] Jundian Song, Shaohua Zhang, Yanxue Gong, and Bingrong Dai. A QoS Evaluation Model for Test-Bed in the Cloud Computing Environment. In *Proceedings of the IEEE Ninth International Conference on e-Business Engineering (ICEBE'12)*, pages 292–295, September 2012. 50

[122] Girish Suryanarayana, Mamadou H. Diallo, Justin R. Erenkrantz, and Richard N. Taylor. Architectural Support for Trust Models in Decentralized Applications. In *Proceedings of the 28th international conference on Software Engineering*, pages 52–61, New York, USA, 2006. ACM Press. 28

[123] Girish Suryanarayana, Mamadou H. Diallo, and Richard N. Taylor. A Generic Framework for Modeling Decentralized Reputation-based Trust Models. Technical Report UCI-ISR-07-4, Institute for Software Research, University of California, Irvine, Aug 2007. 27

[124] M. Swaak, M. Jong, and P. Vries. Effects of information usefulness, visual attractiveness, and usability on web visitors' trust and behavioral intentions. In *Proceedings of the IEEE International Professional Communication Conference (IPCC'09:)*, pages 1–5. IEEE, July 2009. 6

[125] Mohammad Gias Uddin and Mohammad Zulkernine. UMLtrust: towards developing trust-aware software. In *Proceedings of the ACM Symposium on Applied Computing*, SAC '08, pages 831–836. ACM, 2008. 25

[126] Zia ur Rehman, Omar K. Hussain, Sazia Parvin, and Farookh K. Hussain. A Framework for User Feedback Based Cloud Service Monitoring. In *Proceedings of the International Conference on Complex, Intelligent and Software Intensive Systems*, pages 257–262, Los Alamitos, CA, USA, 2012. IEEE Computer Society. 51

[127] Joana Urbano, Ana P. Rocha, and Eugénio Oliveira. *Agreement Technologies*, volume 8 of *Law, Governance and Technology*, chapter A Socio-Cognitive Perspective of Trust, pages 419–429. Springer, 2013. 8

[128] A. Van Lamsweerde. Elaborating security requirements by construction of intentional anti-models. *Proceedings of the 26th International Conference on Software Engineering*, pages 148–157, 2004. 27

[129] Axel van Lamsweerde and Emmanuel Letier. Handling Obstacles in Goal-Oriented Requirements Engineering. *IEEE Transactions on Software Engineering*, 26(10):978–1005, October 2000. 26

[130] Verizon. 2015 Data Breach Investigations Report. Technical report, Verizon, 2015. 2

[131] R. Villarroel, E. Fernández-Medina, and M. Piattini. Secure information systems development - a survey and comparison. *Computers & Security*, 24(4):308–321, 2005. 2

[132] Mark Vinkovits, René Reiners, and Andreas Zimmermann. TrustMUSE: A model-Driven Approach for Trust Management. In *Trust Management VIII: 8th IFI WG 11.11 International Conference on Trust Management*, volume 430 of *IFIP Advances in Information and Communication Technology*, pages 13–27. Springer Berlin Heidelberg, 2014. 29

[133] Quang-Anh Nguyen Vu, Salima Hassas, Frederic Armetta, Benoit Gaudou, and Richard Canal. Combining Trust and Self-Organization for Robust Maintaining of Information Coherence in Disturbed MAS. In *Proceedings of the Fifth IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO'11)*, pages 178–187. IEEE, 2011. 29

[134] Yao Wang and Julita Vassileva. A Review on Trust and Reputation for Web Service Selection. In *Proceedings of the 27th International Conference on Distributed Computing Systems Workshops (ICDCSW'07)*, pages 25–25. IEEE, 2007. 21

[135] Phillip J. Windley, Kevin Tew, and Devlin Daley. A Framework for Building Reputation Systems. `http://www.windley.com/essays/2006/dim2006/framework_for_building_reputation_systems`, 2006. 28

[136] Marianne Winslett, Ting Yu, Kent E. Seamons, Adam Hess, Jared Jacobson, Ryan Jarvis, Bryan Smith, and Lina Yu. Negotiating Trust on the Web. *IEEE Internet Computing*, 6(6):30–37, 2002. 23, 42, 184

[137] Li Xiong and Ling Liu. PeerTrust: Supporting Reputation-Based Trust for Peer-to-Peer Electronic Communities. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):843–857, July 2004. 42, 172, 184, 187

[138] Zheng Yan and Silke Holtmanns. Trust Modeling and Management: from Social Trust to Digital Trust. *Computer Security, Privacy and Politics: Current Issues, Challenges and Solutions*, January 2008. 6, 20, 22, 36, 49

[139] Zheng Yan and C. Prehofer. Autonomic Trust Management for a Component-Based Software System. *IEEE Transactions on Dependable and Secure Computing*, 8(6):810–823, 2011. 30

[140] Zheng Yan, Peng Zhang, and Teemupekka Virtanen. Trust evaluation based security solution in ad hoc networks. In *Proceedings of the Seventh Nordic Workshop on Secure IT Systems, (NordSec'03)*, 2003. 21, 35

[141] Nicola Zannone. *A Requirements Engineering Methodology for Trust, Security, and Privacy*. PhD thesis, University of Trento, Italy, 2007. 80

[142] Ping Zhang, A Durresi, and L Barolli. Survey of Trust Management on Various Networks. In *Proceedings of the International Conference on Complex, Intelligent and Software Intensive Systems (CISIS'11)*, pages 219–226, 2011. 21, 33