Requirements Engineering meets Trust Management*

Model, Methodology, and Reasoning

Paolo Giorgini[1], Fabio Massacci[1], John Mylopoulos[1,2], and Nicola Zannone[1]

[1] Department of Information and Communication Technology
University of Trento - Italy
{massacci,giorgini,zannone}@dit.unitn.it
[2] Department of Computer Science
University of Toronto - Canada
jm@cs.toronto.edu

Abstract. The last years have seen a number of proposals to incorporate Security Engineering into mainstream Software Requirements Engineering. However, capturing trust and security requirements at an organizational level (as opposed to a design level) is still an open problem.
This paper presents a formal framework for modeling and analyzing security and trust requirements. It extends the Tropos methodology, an agent-oriented software engineering methodology. The key intuition is that in modeling security and trust, we need to distinguish between the actors that manipulate resources, accomplish goals or execute tasks, and actors that own the resources or the goals. To analyze an organization and its information systems, we proceed in two steps. First, we built a trust model, determining the trust relationships among actors, and then we give a functional model, where we analyze the actual delegations against the trust model, checking whether an actor that offers a service is authorized to have it.
The formal framework allows for the automatic verification of security and trust requirements by using a suitable delegation logic that can be mechanized within Datalog. To make the discussion more concrete, we illustrate the proposal with a Health Care case study.

keywords : Requirements Engineering for Security and Trust, Agent-Oriented Technologies, Security Engineering, Trust Models for Modeling Business and Organizations

1 Introduction

Trust Management is one of the main challenges in the development of distributed open information systems (IS). Not surprisingly, Security Engineering

* This work has been partially funded by the IST programme of the EU Commission, FET under the IST-2001-37004 WASP project and by the FIRB programme of MIUR under the RBNE0195K5 ASTRO Project. We would like to thank the anonymous reviewers for useful comments.

has received substantial attention in the last years [3, 7, 10]. Looking at traditional approaches to software requirements engineering, we find that security is treated as a non-functional requirement [6] which introduces quality constraints under which the system must operate [24, 26]. Software designers have recognized the need to integrate most non-functional requirements (such as reliability and performance) into the software development processes [8], but security still remains an afterthought. Worse still, trust is often left entirely outside the picture.

This often means that security mechanisms have to be fitted into a pre-existing design which may not be able to accommodate them due to potential conflicts with functional requirements or usability. Moreover, the implementation of the software system may assume trust relationships among users or between users and the system that are simply not there. Alternatively, the implementation may introduce protection mechanisms that just hinder operation in a trusted domain that was not perceived as a trusted domain by the software engineer. In a nutshell, current methodologies for IS development do not resolve security- and trust-related concerns early on [25].

This has spurred a number of researchers to model security and trust requirements into "standard" software engineering methodologies. Jürjens proposes UMLsec [16], an extension of the Unified Modelling Language (UML), for modeling security related features, such as confidentiality and access control. Lodderstedt et al. present a modeling language, based on UML, called SecureUML [21]. Their approach is focused on modeling access control policies and how these (policies) can be integrated into a model-driven software development process. McDermott and Fox adapt use cases [22] to analyze security requirements, by introducing the abuse case model: a specification of complete interaction between a system and one or more actors, where the result of the interaction is harmful to the system, one of the actors, or one of the stakeholders of the system. Guttorm and Opdahl [15] model security by defining the concept of a misuse case as the inverse of a use case, which describes a function that the system should not allow.

One of the major limitations of all these proposals is that they treat security and trust in system-oriented terms, and do not support the modeling and analysis of trust and trust relationships at an organizational level. In other words, they are targeted to *model a computer system* and the policies and access control mechanisms it supports. In contrast, to understand the problem of trust management and security engineering we need to *model* the organization and the relationships between all involved actors, the system being just one possible actor. For instance, Jürjens introduce cryptographic functions which represent a particular implementation of some trust-protection mechanism at the digital level. However, an analysis of operational Health Care systems suggests that (for better or worse) most medical data are still only available in paper form. In such a setting, cryptographic mechanisms are largely irrelevant, whereas physical

locks are very useful in avoiding untrusted access to sensitive medical data[3]. Yet, once we focus on the digital solution, we end up having little room to specify physical protection requirements at the organizational (as opposed to IS) level.

Thus, we need to focus on requirement engineering methodologies that allow for modeling organizations and actors, and enhance these with notions of trust and trust relationships. To this extent, Tropos - an agent-based software engineering methodology [4, 5] are particularly well suited. For example, in [19, 20] Liu et al. have shown how to use Tropos to model privacy and security concerns of an organization. However, in [13] the authors have shown that Tropos lacks the ability to capture at the same time the functional and security features of the organization. In [23] a structured process integrate security and system engineering has been proposed. However, a formal framework for modeling and analyzing security requirements within Tropos is still missing.

In this paper we introduce a process that integrates trust, security and system engineering, using the same concepts and notations used during "traditional" requirements specification. Building upon [23], we propose a solution that is based on augmenting the i*/Tropos framework to take trust into account. The key intuition is to distinguish and make explicit the notion of offering a service and owning a service[4] and the notions of functional dependency and trust dependency. A functional dependency can lead to the delegation of tasks, whereas a trust dependency can lead to the delegation of permissions.

Next (§2) we provide an brief description of the Tropos methodology and introduce a simple Health Care information system that will be used as case study throughout the paper. Then we describe the basic concepts and diagrams that we use for modeling trust (§3), followed by their formalization (§4), and implementation, along with some experimental results (§5). Finally, we conclude the paper with some directions for future work (§6).

2 Case Study

This section presents a simple health care IS to illustrate our approach. Security and trust are key issues for health care information systems, with privacy, integrity and availability of health information being the major security concerns [2].

The Tropos methodology [4, 5] strives to model both the organizational environment of a system and the system itself. It uses the concepts of actor, goal,

[3] For example, the file of a patient waiting for a kidney transplant in a high-profile nephrology center contains many paper documents that are copies of reports from surgeons or clinicians from the referring hospitals of the patient. These documents are by far more sensitive than the patient's date and place of birth or waiting list registration number in the medical information system.

[4] Here it is an example derived from EU privacy legislation: a citizen's personal data is processed by an information system (which offer a data access service) but it is owned by the citizen himself whose consent is necessary for the service to be delivered to 3rd parties.

task, resource and social dependency for defining obligations of actors (dependees) to other actors (dependers). Actors have strategic goals within the system or the organization and represent (social) agents (organizational, human or software), roles etc. A goal represents some strategic interest of an actor. A task represents a way of doing something (in particular, a task can be executed to satisfy a goal). A resource represents a physical or an informational entity. In the rest of the paper, we say service for goal, task, or resource. Finally, a dependency between two actors indicates that one actor depends on another to accomplish a goal, execute a task, or deliver a resource.

We start the Health Care example by considering the following actors:

- *Patient*, that depends on the hospital for receiving appropriate health care;
- *Hospital*, that provides medical treatment and depends on the patients for having their personal information.
- *Clinician*, physician of the hospital that provides medical health advice and, whenever needed, provide accurate medical treatment;
- *Health Care Authority* (HCA) that control and guarantee the fair resources allocation and a good quality of the delivered services.

Figure 1 shows the dependency model among these actors. Actors are represented as circles; dependums - goals, tasks and resources - are respectively represented as ovals, hexagons and rectangles; and dependencies have the form *depender* $\rightarrow$ *dependum* $\rightarrow$ *dependee*. The *Patient* depends on the *Hospital* for receiving medical treatments, and in turn, the *Hospital* depends on the *Clinician* for providing such treatments. *Clinician* depends on *Patients* for their personal information and on the *Hospital* for specific professional consultancies and for patient personal information. The *Hospital* depends on other *Clinicians* for providing professional consultancies and on *HCA* for checking equity resource distribution. Finally, *HCA* depends on *Patient* for personal information.

Finally we introduce the *Medical Information System* as another actor who, according the current privacy legislation, can share patient medical data if and only if consent is obtained from the patient in question. The *Medical Information System* manages patients information, including information about the medical treatments they have received. Figure 2 shows the final dependency model.

3 Security-Aware Tropos

The Tropos models so far say nothing about security requirements. Loosely speaking, the dependee is a server and the depender is a client. There is an implicit trust and delegation relationship between the two. In our extended modeling framework, we identify four relationships:

trust (among two agents and a service), so that A trust B on a certain goal G;
delegation (among two agents and a service), whenever A explicitly delegates to B a goal, or the permission to execute a task or access a resource;
offer (between an agent and a service), so that A can offer to other agents the possibility of fulfilling a goal, executing a task or delivering a resource;

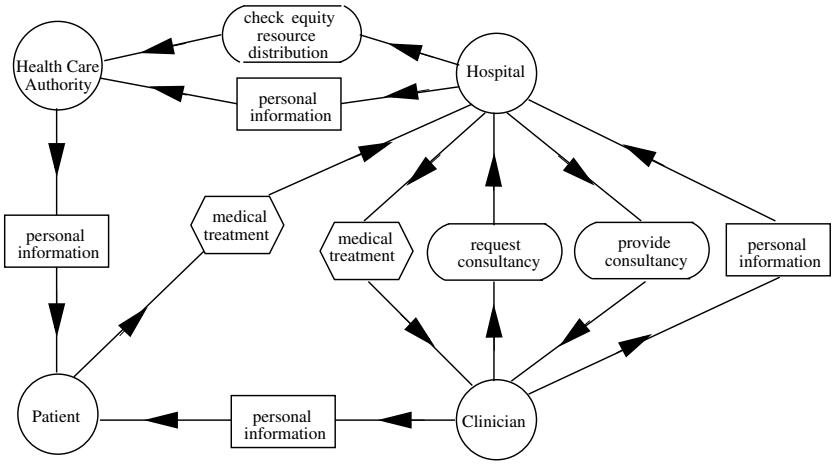


Fig. 1. The first Health Care System dependency model (without the Medical Information System actor)

ownership (between an agent and a service), whenever an agent is the legitemate owner of a goal, task or resource.

Note the difference between owning a service and offering a service. For example, a patient is the legitemate owner of his personal data. However the data may be stored on a Medical Information System that offers access to the data. This distinction explains clearly why IS managers need the consent of the patient for data processing. Also note the difference between trust and delegation. Delegation marks a formal passage in the requirements modeling: a TM certificate will have to be eventually issued for the delegatee when implementing the system. Such certificate needs not to be digital, but it marks presence of a transaction. In contrast, trust marks simply a social relationship that is not formalized by a "contract" (such as digital credential). There might be cases (e.g. because it is impractical or too costly), where we might be happy with a "social" protection, and other cases in which security is essential. Such decision must be taken by the designer and the formal model just offers support to spot inconsistencies. The basic effect of delegation is augmenting the number of permission holders.

Intuitively, we have split the trust and delegation aspects of the dependency relation. Moreover, we do not assume that a delegation implies a trust. Using this extension of the modeling framework, we can now refine the methodology:

1. design a trust model among the actors of the systems;
2. identify who owns goals, tasks, or resources and who is able to fulfill goals, execute tasks or deliver resources;
3. define functional dependencies and delegations of goals among agents building a functional model.

The basic idea is that the owner of an object has full authority concerning access and disposition of his object, and he can also delegate it to other actors.

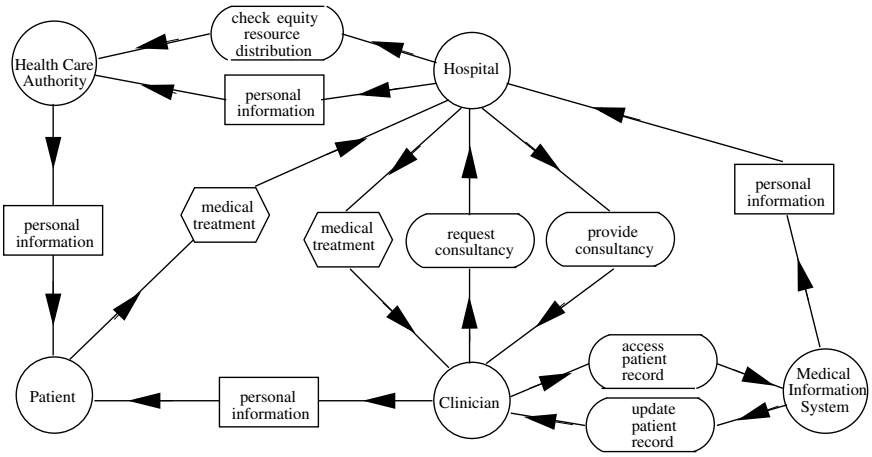
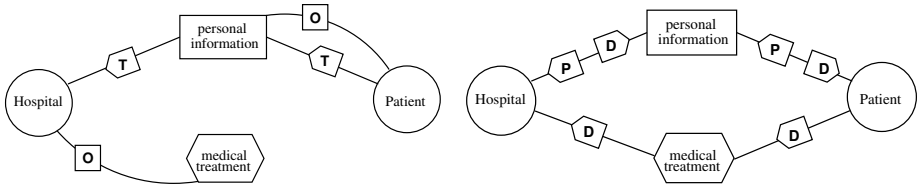


Fig. 2. The final Health Care System dependency model (with the Medical Information System actor)



(a) Trust Model (b) Functional Model

Fig. 3. Patient-Hospital Basic Dependencies

We represent this relationship as an edge labelled by **O**. We use trust (**T**) to model the basic trust relationship between agents and permission (**P**) to model the actual transfer of rights in some form (e.g. a digital certificate, a signed paper, etc.), and **D** for a Tropos dependency. There are other relations in Tropos, but we do not use them here.

The new constructs and the methodology make it possible to analyze the trust relationship between actors and the consequent integrated security and functional requirements. Figure 3-a and Figure 3-b show, respectively, the trust model and the functional model with just *Patient* and *Hospital*, as a first modeling attempt. Here, the *Hospital* owns medical treatments, the *Patient* owns his own personal information and trusts the *Hospital* for his personal data. In the functional model, *Patient* depends on *Hospital* for medical treatments. Since *Hospital* needs personal information to provide accurate medical treatment, *Patient* permits the use of his personal information to *Hospital*.

We refine the system building the trust model (Figure 4) corresponding to the original Tropos model of Figure 1. *Clinician* owns medical treatments. *Patient*

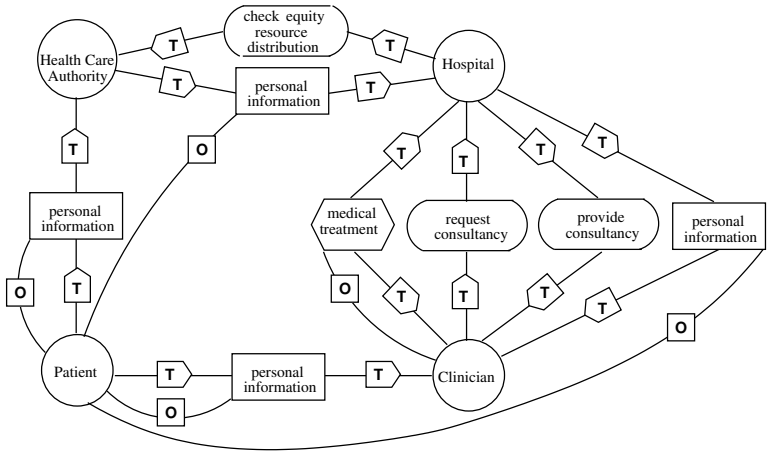


Fig. 4. Health Care System-2 trust model

trusts *HCA* and *Clinician* for his personal information, and *HCA* trusts *Hospital* for it. Further, *Hospital* trusts *HCA* for checking equity resource distribution. *Clinician* trusts *Hospital* for medical treatment and for requesting specific professional consulting, and *Hospital* trusts *Clinician* for providing such consulting and for patient personal information. Notice at top of Figure 4 that there is a trust relationship between two actors (HCA and Hospital) on a resource that is owned by neither of them.

The next step is to add the *Medical Information System* and its relationship with other actors. Figure 5 and Figure 6 corresponding to the dependencies model in Figure 2, show respectively the trust model and the functional model. In the trust model we consider the trust relationship between *Hospital* and *Medical System Information* for patient personal information, and in the functional model the dependency between *Clinician* and *Medical Information System* to access patient record and to update patient record.

An interesting feature of Tropos is the refinement analysis and the usage of rationale diagrams that explain relationships among actors. Specifically, the goal of accessing a patient record introduced in Figure 2, can be and-decomposed in three subgoals: request patient personal data, check authorization and send medical information. To save on space, we merge the trust model and the functional model for the rationale diagram in Figure 7. We can see that after *Medical Information System* requests patient personal information to *Clinician*, it requests also an authorization to send patient medical information to *Clinician*. It can get it directly by the *Patient* or by the *Clinician* through delegation.

4 Formalization

In the "trust-management" approach to distributed authorization, a "requester" submits a request, possibly supported by a set of "credentials" issued by other

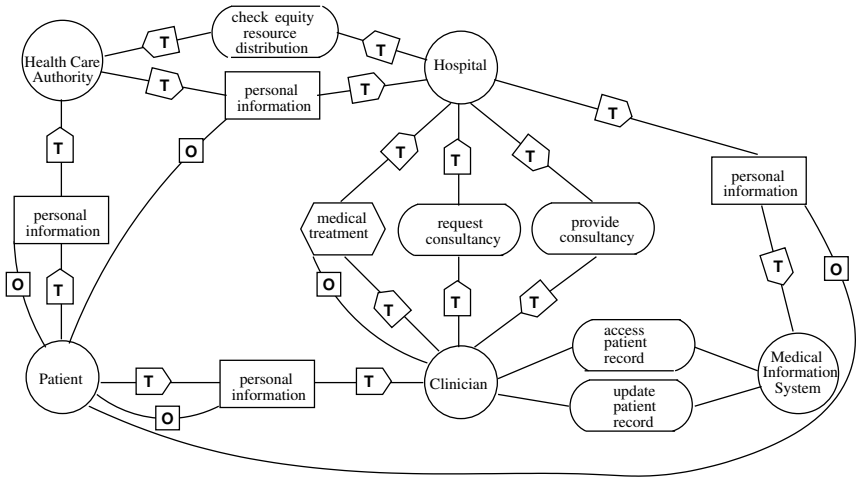


Fig. 5. Health Care System-3 trust model

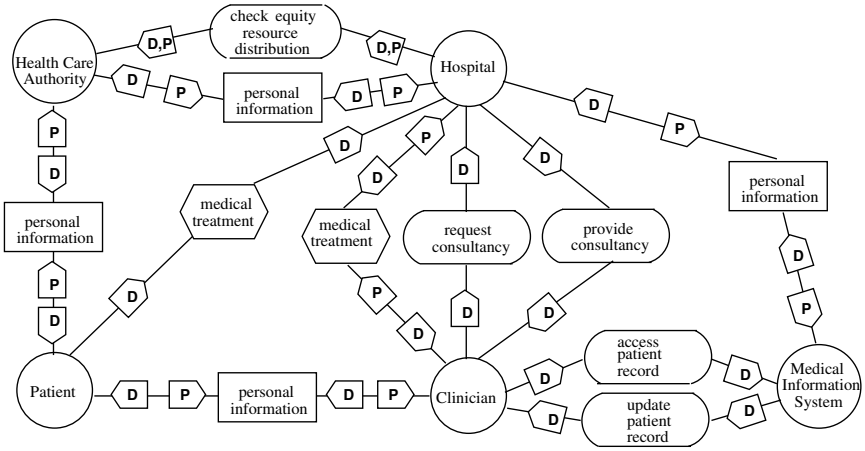


Fig. 6. Health Care System-3 functional model

parties, to an "authorizer", who controls the requested resources. To this end, we consider some features of delegation logics to model security requirements. Particularly, we follow Li et al [18] that provides a logical framework for representing security policies and credentials for authorization in large-scale, open, distributed systems. To simplify authorization in a decentralized environment, Li, Grosof and Feigenbaum use a system where access-control decisions are based on authenticated attributes of the subjects, and attribute authority is decentralized. They then develop a logic-based language, called *Delegation Logic* (DL) [17], to represent policies, credentials, and requests in distributed authorization that satisfy the above requirements. Note that they use the term *authorization* to denote the process of "authentication + access control".

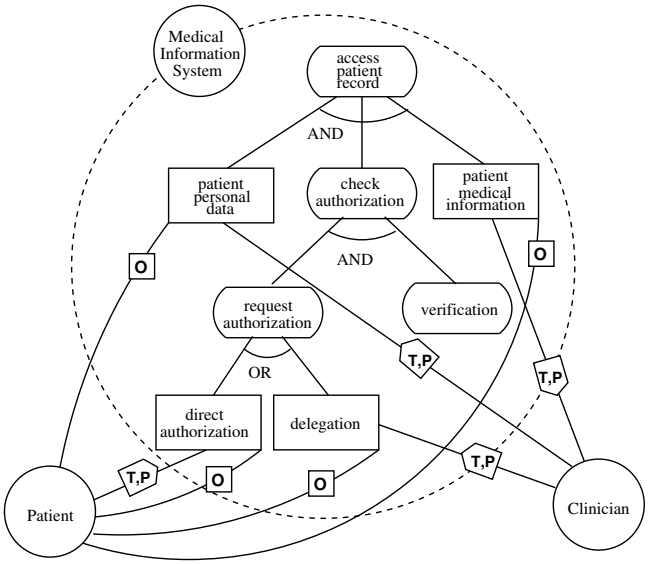


Fig. 7. Rationale Diagram

Formal Tropos	**Secure Tropos**
fulfilled(`Service` : s)	owns(`Actor` : a, `Service` : s) has(`Actor` : a, `Service` : s) offers(`Actor` : a, `Service` : s) fulfills(`Actor` : a, `Service` : s)

Table 1. Properties for the single agent

At first we introduce the predicates used for modeling properties of an actor (Table 1) and relationships between actors (Table 2). In defining these predicates, we don't distinguish between goals, tasks and resources, and treat them all as services, instead. Thus, we say "fulfill a service" for "accomplish a goal", "execute a task", or "deliver a resource". The intuition behind predicate *owns* is that owns(a, s) holds if the agent a owns the service s. The owner of a service has full authority concerning access and disposition of his service, and he can also delegate this authority to other actors. The basic idea of has is that when someone has a service, he has authority concerning access and disposition of the service, and he can also delegate this authority to other actors if the owner of the service agrees. When an actor has the capabilities to fulfill a service, he offers it. This means that offers(a, s) holds if a offers s. We assume that a can offer the service if he has it. The predicates fulfilled and fulfills are true when the service are fulfilled by an actor. Particularly, predicate fulfills(a, s) holds if actor a fulfills the service s, and predicate fulfilled(s) holds if s has been fulfilled. Formal Tropos already includes the predicate fulfilled [12].

Formal Tropos
$\mathsf{depends}(\texttt{Actor} : a, \texttt{Service} : s, \texttt{Actor} : b)$
Secure Tropos
$\mathsf{trustBL}\big(\texttt{Actor} : a, \texttt{Service} : s, \texttt{Actor} : b, \mathcal{N}^+ \cup \{*\} : n, \texttt{ActorSet} : \mathcal{B}\big)$
$\mathsf{delegBL}\big(\texttt{id} : idC, \texttt{Actor} : a, \texttt{Service} : s, \texttt{Actor} : b, \mathcal{N}^+ \cup \{*\} : n, \texttt{ActorSet} : \mathcal{B}\big)$

Table 2. Relationship between actors

Example 1. The patient owns his data and he has full authority concerning its access and disposition. In particular, the owner of the service has the service. In our framework we model these notions, respectively, as $\mathsf{owns}(patient1, record1)$ and $\mathsf{has}(patient1, record1)$.

Example 2. Once the Health Care Authority has the patient records and the hospital gives it the goal to check behavior of patients and of the doctors, the HCA offers the goal and then fulfills it. Following we show as we model this in Secure Tropos $\mathsf{offers}(hca, check)$ and $\mathsf{fulfills}(hca, check)$.

As for trust, we present predicate $\mathsf{trustBL}$: $\mathsf{trustBL}(a, s, b, n, \mathcal{B})$ holds is actor a trusts actor b for service s; n is called the *trust depth* ("*" means unlimited depth); and $\mathcal{B}$ is called *black list*. As suggest by Li et al. [17] for their delegation logics, trust has depth, which is either a positive integer or "*" ("*" means unlimited depth). One way to view trust depth is the number of re-delegation of permission steps that are allowed, where depth 1 means that no re-delegation of permission is allowed, depth 2 means that one further step is allowed, depth 3 means that two further steps are allowed, and depth * means that unlimited re-delegation of permission is allowed. The black list is the set that the actor a distrusts at least for what concerns this permission. $\mathsf{delegBL}(idC, a, s, b, n, \mathcal{B})$ holds is actor a delegates the service s to actor b. The actor a is called the *delegater*; the actor b is called the *delegatee*; idC is the certificate identifier; n is the *delegation depth*; and $\mathcal{B}$ is called *black list*. The latter represents the set of actors that the delegater doesn't want to have the object. The idea behind black-lists in trust and delegation is modeling exceptions along the chain of trust. For example, a patient may want to delegate the permission to read his personal data to his general practitioner and to all agents trusted by him (delegation with depth 1). However, he may want to restrict such blank transfer of rights to avoid that the information goes to somebody he distrusts (e.g. his previous general practitioner). A delegation has depth, as for trust. We can also define an abbreviation for a delegation chain as

$$\mathsf{delegBLChain}(a, o, b) \equiv \begin{cases} \exists k \;\text{ s.t. }\; \exists a_1 \ldots a_k \;\; \exists n_1 \ldots n_{k-1} \;\; \exists \mathcal{B}_1 \ldots \mathcal{B}_{k-1} \\ \forall i \in [1 \ldots k-1]\; \mathsf{delegBL}(id_i, a_i, o, a_{i+1}, n_i, \mathcal{B}_i) \;\wedge \\ a_1 = a \wedge a_k = b \end{cases}$$

Example 3. Patient trusts Clinician on his medical data.

$$\mathsf{trustBL}(patient1, record1, clinician1, 1, \emptyset)$$

Trust model	
Ax1:	$\mathsf{has}(A,S) \leftarrow \mathsf{owns}(A,S)$
Ax2:	$\mathsf{trustBL}(A,S,B,N-1,\mathcal{B}) \leftarrow \mathsf{trustBL}(A,S,B,N,\mathcal{B}) \wedge N > 2$
Ax3:	$\mathsf{trustBL}(A,S,C,P,\mathcal{B}_1 \cup \mathcal{B}_2) \leftarrow \mathsf{trustBL}(A,S,B,N,\mathcal{B}_1) \wedge$ $\mathsf{trustBL}(B,S,C,M,\mathcal{B}_2) \wedge$ $N > 2 \wedge P = min\{N-1, M\}$
Functional model	
Ax4:	$\mathsf{has}(B,S) \leftarrow \mathsf{delegBL}(ID,A,S,B,N,\mathcal{B})$
Ax5:	$\mathsf{fulfilled}(S) \leftarrow \mathsf{fulfills}(A,S)$
Ax6:	$\mathsf{fulfills}(A,S) \leftarrow \mathsf{has}(A,S) \wedge \mathsf{offers}(A,S)$

Table 3. Axioms for trust model and functional model

When the Clinician visits his patient he requests to the Medical Information System the patient record. The Medical Information System delegates patient record to the patient's clinician. The clinician cannot delegate the record to others actors. Formally this is $\mathsf{delegBL}(m1, medicalIS, record1, clinician1, 1, \emptyset)$.

In Table 3 we present the axiom for the trust model and for the functional model. As mentioned earlier, the owner of a service has full authority concerning access and disposition of it. Thus, Ax1 states that if an actor owns a service, he has it. Ax2 states that if someone trusts with depth N, then he also trusts with smaller depth. Ax3 describes the trust relationship, i.e, it completes the trust relationship between actors. Ax4 says that a delegatee has the service he was delegated. Ax5 states that an actor fulfills a service, then the service is (eventually) fulfilled. Ax6 states that if an actor has a service and offers it, then he (eventually) fulfills it.

Properties are different from axioms: they are constraints that must be checked. It is up to designer to choose which properties his own design should respect. If the set of constraints is not consistent, i.e. they cannot all be simultaneously satisfied, the system is inconsistent, and hence it is not secure. In Table 4 we use the $A \Rightarrow? B$ to mean that one must check that each time A holds, it is desirable that B also holds. Pro1 and Pro2 state that if an agent offers or delegates, he should have the object. Pro3 says that to fulfill a goal an actor must be able to use and offer it. Pro4, Pro5 and Prop6 state that if an actor has, offers, or fulfills a goal and this goal belongs to another actor, the last has to trust the first one. Pro7, Pro8 are used to verify whether the delegatee is not in the black list. Pro9 and Pro10 state that an actor who delegates something to an other, has to trust him. Rights or privileges can be given to trusted agents that are then accountable for the agents to whom may further delegate this right to. So the agents should only delegate to agents that they trust. This forms a delegation chain. If any agent along this chain fails to meet the requirements associated with a delegated right, the chain is broken and all agents following the failure are not permitted to perform the action associated with the right. Thus, Prop11 is used to verify if the delegate chain is valid.

Pro1:	$\mathsf{offers}(A,S) \Rightarrow?\ \mathsf{has}(A,S)$
Pro2:	$\mathsf{delegBL}(ID,A,S,B,N,\mathcal{B}) \Rightarrow?\ \mathsf{has}(A,S)$
Pro3:	$\mathsf{fulfills}(A,S) \Rightarrow?\ \mathsf{offers}(A,S)$
Pro4:	$\mathsf{has}(B,S) \wedge \mathsf{owns}(A,S) \Rightarrow?\ \exists N\ \exists\mathcal{B}\ \mathsf{trustBL}(A,S,B,N,\mathcal{B})$
Pro5:	$\mathsf{offers}(B,S) \wedge \mathsf{owns}(A,S) \Rightarrow?\ \exists N\ \exists\mathcal{B}\ \mathsf{trustBL}(A,S,B,N,\mathcal{B})$
Pro6:	$\mathsf{fulfills}(B,S) \wedge \mathsf{owns}(A,S) \Rightarrow?\ \exists N\ \exists\mathcal{B}\ \mathsf{trustBL}(A,S,B,N,\mathcal{B})$
Pro7:	$\mathsf{delegBL}(ID,A,S,B,N,\mathcal{B}) \wedge \mathsf{owns}(A,S) \Rightarrow?\ \forall X \in \mathcal{B}\ \neg\mathsf{has}(X,S)$
Pro8:	$\mathsf{delegBL}(ID,A,S,B,N,\mathcal{B}) \Rightarrow?\ B \notin \mathcal{B}$
Pro9:	$\mathsf{delegBL}(ID,A,S,B,N,\mathcal{B}_1) \Rightarrow?\ \exists M \geq N\ \exists\mathcal{B}_2\ \mathsf{trustBL}(A,S,B,M,\mathcal{B}_2) \wedge B \notin \mathcal{B}_1 \cup \mathcal{B}_2$
Pro10:	$\mathsf{delegBLChain}(A,S,B) \Rightarrow?\ \exists N\ \exists\mathcal{B}\ \mathsf{trustBL}(A,S,B,N,\mathcal{B}) \wedge B \notin \mathcal{B}$ $\exists M\ \ \exists A_1 \ldots A_M\ \ \exists N_1 \ldots N_{M-1}\ \ \exists\mathcal{B}_1 \ldots \mathcal{B}_{M-1}$
Pro11:	$\mathsf{delegBLChain}(A,S,B) \Rightarrow?\ \forall i \in [1 \ldots M-1]\ \mathsf{delegBL}(ID_i,A_i,S,A_{i+1},N_i,\mathcal{B}_i) \wedge$ $A_1 = A \wedge A_M = B \wedge N_i > N_{i+1} \wedge \mathcal{B}_i \subseteq \mathcal{B}_{i+1} \wedge A_{i+1} \notin \mathcal{B}_i$

Table 4. Desirable Properties of a Design

```
has(A,S) :- owns(A,S).
has(B,S) :- delegate(ID,A,S,B,N).
fulfill(A,S) :- has(A,S), offer(A,S).
fulfilled(S) :- fulfill(A,S).
trustBL(A,S,B,N) :- #succ(N,M), trustBL(A,S,B,M), N>0.
trustBL(A,S,C,P) :- -bL(C), #succ(P,N), trustBL(A,S,B,N),
                    trustBL(B,S,C,M), M>=N, N>1.
trustBL(A,S,C,M) :- -bL(C), trustBL(A,S,B,N), trustBL(B,S,C,M), N>M, N>1.
```

Table 5. Axioms in Datalog

There are additional properties that we have not listed due to a lack of space, such as checking delegation to actors that cannot have a service directly.

5 Implementation and Experimental Results

In order to illustrate our approach we formalize the case study and check-model it in Datalog [1]. A datalog logic program is a set of rules of the form L:-$L_1 \wedge ... \wedge L_n$ where L, called head, is a positive literal and $L_1, ..., L_n$ are literals and they are called body. Intuitively, if $L_1, ..., L_n$ are true in the model then L must be true in the model. The definition can be recursive, so defined relations can also occur in bodies of rules. Axioms of the form $A \leftarrow B \wedge C$ can be represented as $A\ :-\ B,\ C$. In Datalog properties can be represented as the constraint $:-A,\ \text{not}\ B$.

We use the DLV system [9] for the actual analysis. Consistency checks are standard checks to guarantee that the security specification is not self-contradictory. Inconsistent specifications are due to unexpected interactions among constraints in the specifications. The consistency checks are performed automatically by DLV. The simplest consistency check verifies whether there is any valid scenario that respects all the constraints of the security specification.

```
:- offer(A,S), not has(A,S).
:- delegate(ID,A,S,B,N), not has(A,S).
:- offer(B,S), owns(A,S), not trustNP(A,S,B), A<>B.
:- fulfill(B,S), owns(A,S), not trustNP(A,S,B), A<>B.
:- delegateChain(A,S,C,N), not trustBL(A,S,C,N).
```

Table 6. Some properties in Datalog

```
trustFull(Pat,Rec,X) :- isHCA(X), owns(Pat,Rec).
trust(Pat,Rec,Cli,1) :- isClinicianOf(Cli,Pat), owns(Pat,Rec).
trustFull(hca,Rec,hospital) :- isRecord(Rec).
trustFull(hospital,Rec,mIS) :- isRecord(Rec).
trustFull(hospital,Rec,X) :- isClinician(X), isRecord(Rec).
```

Table 7. Health Care System-3 trust relationship in Datalog

Example 4. For model checking purposes we consider two patients, three clinicians, and one HCA. Patients trust completely the HCA for their personal information. Then we rapresent the relation between Patient and Clinician shown in Figure 4, that is, the Patient trusts his clinicians with depth 1. Further, HCA trusts completely Hospital for patients personal informations. Finally, we present the relationship between Hospital and Clinician on patient personal information. Below we introduce the constraint to verify whether only the clinicians of the patient can have patient information.

```
:- trust(Pat,Rec,Cli,N), owns(Pat,Rec), isClinician(Cli),
   not isClinicianOf(Cli,Pat).
```

The DLV system reports an inconsistency since all Clinicians are authorized to have the personal information of any patient. Ideally we would authorize only the clinician of the patient to have patient data.

Example 5. The trust relationship among actor in Figure 6 and in Figure 7 is formalized in Table 7 and is described below:

1. Patient trusts completely HCA and he trusts directly his Clinician,
2. HCA trusts completely Hospital,
3. Hospital trusts completely Medical Information System, and
4. Medical Information System trusts completely Clinicians.

We can check whether only the clinicians of the patient can have patient personal information according to Example 4. The DLV system report an inconsistency: in the current design every Clinician is implicitly authorized to have patient personal information. To resolve this problem, we have to change the trust model using the following trust relation between the Medical Information System and the Clinician.

```
trust(mIS,Rec,Cli,1) :- isClinicianOf(Cli,Pat),owns(Pat,Rec).
```

In other words, the *Medical Information System* allows an actor to access directly the records of a patient if the actor is the physician of the patient.

We can now analyze the complete trust and functional model. In particular, we check whether the delegater trusts the delegatee. The refined result is that patient's consent must be sought for any other agent such as clinician's colleagues to be able to access at patient medical information, and the patient must be notified of every access. So the clinician has to request a consulting to colleagues through the hospital and the patient must give the permission to access the data.

It is also possible to make additional queries aimed at verifying a number of security principles such as least-privilege, or need-to-know policies as done by Liu et al. [20] in their security requirements model formalized in Alloy.

6 Conclusions

The main contribution of this paper is the introduction of a formal model and a methodology for analyzing trust during early requirement engineering. To this end, we have proposed an enhancement of Tropos that is based on the clear separation of functional dependencies, trust and delegation relationships. This distinction makes it possible to capture organization-oriented security and trust requirements without being caught into the technical details about how these will be realized through digital certificates or access control mechanisms. The modeling process we envision has the advantage of making clear why and where trust management and delegation mechanisms are necessary, and which trust relationships or requirements they address.

The framework we proposed supports the automatic verification of security requirements and trust relationships against functional dependencies specified in a formal modeling language. The model can be easily modified to account for degrees of trust. Levels of trust can be captured by using a qualitative theory for goal analysis. See [14] for details.

Plans for future work include adding time to trust models and analyzing these new features with the Formal Tropos T-Tool [11].

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. R. Anderson. A security policy model for clinical information systems. In *Proc. of the 15th IEEE Symp. on Security and Privacy*. IEEE Comp. Society Press, 1996.
3. R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Computer Publishing, 2001.
4. P. Bresciani, F. Giunchiglia, J. Mylopoulos, and A. Perini. TROPOS: An Agent-Oriented Software Development Methodology. *JAAMAS*, (To appear).
5. J. Castro, M. Kolp, and J. Mylopoulos. Towards Requirements-Driven Information Systems Engineering: The Tropos Project. *Inform. Sys.*, 27(6):365–389, 2002.

6. L. Chung and B. Nixon. Dealing with non-functional requirements: Three experimental studies of a process-oriented approach. In *Proc. of ICSE'95*, 1995.
7. R. Crook, D. Ince, L. Lin, and B. Nuseibeh. Security requirements engineering: When anti-requirements hit the fan. In *Proc. of RE'02*. IEEE Computer Society, 2002.
8. A. Dardenne, A. V. Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 1991.
9. T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in dlv. In *Proc. of IJCAI'03*. Morgan Kaufmann Publishers, 2003.
10. P. T. Devanbu and S. G. Stubblebine. Software engineering for security: a roadmap. In *ICSE - Future of SE Track*, pages 227–239, 2000.
11. A. Fuxman, L. Liu, M. Pistore, M. Roveri, and J. Mylopoulos. Specifying and analyzing early requirements: Some experimental results. In *Proc. of ICRE'03*, page 105. IEEE Computer Society, 2003.
12. A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. Model checking early requirements specifications in tropos. In *Proc. of RE'01*, pages 174–181, Toronto, August 2001. IEEE Computer Society.
13. P. Giorgini, F. Massacci, and J. Mylopoulos. Requirement Engineering meets Security: A Case Study on Modelling Secure Electronic Transactions by VISA and Mastercard. In *Proc. of ER'03*, Chicago, Illinois, 13-16 October 2003.
14. P. Giorgini, E. Nicchiarelli, J. Mylopoulous, and R. Sebastiani. Formal reasoning techniques for goal models. *J. of Data Semantics*, 1, 2003.
15. S. Guttorm. Eliciting security requirements by misuse cases. In *Proceedings of TOOLS Pacific 2000*, 2000.
16. Jan Jürjens. Towards Secure Systems Development with UMLsec. In *Proc. of FASE'01*, 2001.
17. N. Li, B. N. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM TISSEC 03*, 6(1):128–171, 2003.
18. N. Li, W. H. Winsborough, and J. C. Mitchell. Beyond proof-of-compliance: Safety and availability analysis in trust management. In *Proc. of Symposium on Security and Privacy*, 2003.
19. L. Liu, E. Yu, and J. Mylopoulos. Analyzing security requirements as relationships among strategic actors. In *Proc. of SREIS'02*, North Carolina, 2002. Raleigh.
20. L. Liu, E. Yu, and J. Mylopoulos. Security and privacy requirements analysis within a social setting. In *Proc. of RE'03*, pages 151–161, 2003.
21. T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In J.-M. Jezequel, H. Hussmann, and S. Cook, editors, *Proc. of UML'02*, volume 2460, pages 426–441. Springer, 2002.
22. J. McDermott and C. Fox. Using abuse care models for security requirements analysis. In *Proc. of ACSAC'99*, December 1999.
23. H. Mouratidis, P. Giorgini, and G. Manson. Modelling secure multiagent systems. In *Proc. of AAMAS'03*, pages 859–866. ACM Press, 2003.
24. I. Sommerville. *Software Engineering*. Addison-Wesley, 2001.
25. T. Tryfonas, E. Kiountouzis, and A. Poulymenakou. Embedding security practices in contemporary information systems development approaches. *Information Management and Computer Security*, 9:183–197, 2001.
26. E. Yu and L. Cysneiros. Designing for privacy and other competing requirements. In *Proc. of SREIS'02*, North Carolina, 2002. Raleigh.