Int. J. Inf. Secur. (2006) 5:257–274
DOI 10.1007/s10207-006-0005-7

REGULAR CONTRIBUTION

Requirements engineering for trust management: model, methodology, and reasoning

Paolo Giorgini · Fabio Massacci · John Mylopoulos · Nicola Zannone

Published online: 16 August 2006
© Springer-Verlag 2006

Abstract A number of recent proposals aim to incorporate security engineering into mainstream software engineering. Yet, capturing trust and security requirements at an organizational level, as opposed to an IT system level, and mapping these into security and trust management policies is still an open problem. This paper proposes a set of concepts founded on the notions of ownership, permission, and trust and intended for requirements modeling. It also extends Tropos, an agent-oriented software engineering methodology, to support security requirements engineering. These concepts are formalized and are shown to support the automatic verification of security and trust requirements using Datalog. To make the discussion more concrete, we illustrate the proposal with a Health Care case study.

Keywords Requirements Engineering · Agent-oriented software · Security Engineering · Trust models for business and organizations · Verification and validation of software · Privilege management

This work is an expanded and revised version of [19,20].

P. Giorgini · F. Massacci · J. Mylopoulos · N. Zannone (✉)
Department of Information and Communication Technology, University of Trento, Trento, Italy
e-mail: zannone@dit.unitn.it

P. Giorgini
e-mail: giorgini@dit.unitn.it

F. Massacci
e-mail: Fabio.Massacci@unitn.it

J. Mylopoulos
e-mail: jm@dit.unitn.it

1 Introduction

Modern IT systems (such as health care, banking, industry and military systems) store and manipulate large amounts of sensitive data in a distributed setting giving birth to two major security research areas: Trust Management (hereafter TM) and Privilege Management Infrastructure (hereafter PMI) [8,12,14,17,23,27]. To protect and regulate access to such widely distributed data, a number of advanced policy languages, algorithms, and systems for managing security credentials have been developed in the last few years. However, if we look at the connection between these credential-based systems and the requirements of an entire IT system we find a large gap. Managing high-level user requirements is a key issue for the successful and cost-effective development of IT systems [16], but managing security requirements is almost completely ignored [30]. We are interested in answering the question: can we define a disciplined methodology to support IT designers while capturing of high-level trust and privilege management requirements and their implementation?

Working towards an answer, we note that we cannot simply rely on software engineering methodologies. While there are structured processes for capturing high-level functional requirements and detailing them down to system implementation, no such processes exist for TM and PMI. Traditional approaches to software requirements engineering, treat security as a non-functional requirement which introduces quality constraints under which the system must operate [13,37,41]. In software engineering practice, the inclusion of security features within a system is usually done after functional design [39]. This is a critical problem, mainly because security mechanisms have to be fitted into a pre-existing

design which may not be able to accommodate them or might be even in conflict with them.

This has spurred a number of researchers to model security requirements and integrate these models with “standard” software engineering methodologies. For example, Jürjens [24] proposes UMLsec , an extension of the Unified Modeling Language (UML), for modeling security -related features such as confidentiality and access control. Basin et al. [7] focus on modeling RBAC policies and how integrating these (policies) into a model-driven software development process whose main task is the secure configuration of systems. One of the major limitations of all these proposals is that they treat security in system-oriented terms. In other words, they are intended to *model a computer system*, along with its policies and security mechanisms it supports.

Moreover, trust is often left entirely outside the picture. Trust is an important aspect for making decisions on security and particularly influences the specification of TM policies at organizational level (a focus of our paper). Trust is related to belief in honesty, trustfulness, competence, and reliability [9,11,34] and it is a fundamental concept in human behavior. Trust is used to build collaboration between humans and organizations since it is a necessary antecedent for cooperation [5]. Trust leads to constructive and cooperative behavior fundamental for long-term relationships [6]. Over the past 10 years, a number of trust policy languages have been developed to model trust in distributed systems [8,12,17,23,27]. However, there is still a gap between these frameworks and the requirements analysis process. If we do not consider trust in the design, security measures imposed on the system might be heavier-handed than necessary: the implementation may introduce pointless protection mechanisms that just hinder operations in a trusted domain that was not perceived like that by system designers. On the other hand, the implementation of the IT system may implicitly assume trust relationships among users or between users and the system that are simply not there.

To understand the problem of trust management and security engineering we need to *model social and organizational settings*, in terms of relationships between relevant actors, including the system-to-be. Modeling only the digital protection mechanism is not satisfactory. For instance, Jürjens [24] introduces security tagging which represents the need of a particular implementation of some security-protection mechanism when messages are exchanged in a UML collaboration diagram. Yet, an analysis of the security requirements of a real health care authority suggests that (for better or worse) most medical data are still only available in paper format. In such a setting, cryptographic mechanisms are less critical then physical locks in avoiding unauthorized access to sensitive medical data[1]. Thus, by focusing only on digital solutions, we end up having little room to specify physical protection requirements at the organizational (as opposed to software) level. On the contrary, analyzing the structure of organizations *and* the system allows designers to understand which information should be protected and, depending on the nature of the information itself, identify which security mechanisms better protect it.

Thus, we need to focus on requirements engineering methodologies that allow for modeling organizations and actors, and enhance these with notions of trust and permission in order to build a PMI/TM implementation that takes into account *both* the system and the organization. To this extent, Tropos – an agent-based software engineering methodology [10] – is particularly well suited to model both an organizational setting and a system within it. However, in [18] the authors have shown that Tropos lacks the ability to capture at the same time the functional and security features of the organization.

In this paper we introduce a process that integrates trust and security facets into system engineering by using the same concepts and notations used during “traditional” requirements specification. Building upon [18], we propose a solution based on augmenting the i*/Tropos framework to take entitlement, trust, and delegation of permission into account. Our goal is to introduce a PMI/TM into the requirements engineering framework. Essentially, we would like to avoid designing an entire IT system and then retrofitting a PMI/TM on top, when it is already too late to make a proper fit.

The first intuition is that in modeling security and trust, we need to distinguish between actors that want access to a resource, fulfillment of a goal or execution of a plan, from actors that have the capabilities to do any of the above, and – last but not least – actors that are entitled to do any of the above. We call our proposal ECO model for *Entitlements*, *Capabilities*, and *Objectives*.

The second point is to distinguish among functional dependency, trust, and delegation relationships. Intuitively, a functional dependency describes an “agreement” between actors for delivering a resource, fulfilling a goal or executing a plan. If one uses only functional dependencies, he would end up in a traditional RE func-

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

tional model of the organization and the software systems. A trust relation describes the belief of an actor about the behavior of another actor, namely that the trustee will not misuse the resource or task expected from him. For example, we trust the cleaning lady for fulfilling the goal of cleaning the room and doing nothing else (e.g., not snooping among the papers). A delegation describes the formal passage of permission between two actors, which may be done electronically (e.g., a digital certificates), in print (e.g., a signed letter or a issued regulation), or even orally (e.g., an order or an agreed verbal contract).

To devise the PMI/TM structure, we propose to proceed in three steps. First, we build a functional requirements model where we show functional dependencies among actors. We then develop a trust requirements model to study whether trust relations among actors correspond to security requirements. Finally, we built a PMI/TM implementation where the designer can define credential and delegation certificates comparing them with the relations captured in the other models and check whether every actor that provides a service is authorized to do it.

Once conceptual models are developed, we translate them into a formal specification. This allows our framework to support the automatic verification of high-level functional, trust, and security requirements by using a suitable delegation logic that can be mechanized within Datalog [1] and in particular within DLV[2] system [25]. Essentially, we want to guarantee that functional, trust, and security requirements are not contradictory. Finally, we propose an algorithm to map the PMI/TM implementation of our framework into the Role-based Trust-management (hereafter RT) framework [27].

In the next section (Sect. 2) we discuss related work. Then, we introduce a simple health care information system that is used as a running example throughout the paper (Sect. 3). We provide a brief description of the Tropos methodology and describe the basic concepts and diagrams that we use for modeling security (Sect. 4). We show how the running example can be modeled in our framework (Sect. 5). We then present a formalization of the security concepts (Sect. 6) and analyze the running example using the formal framework (Sect. 7). Next, we introduce negative authorizations (Sect. 8), describe the implementation of trust into the RT framework (Sect. 9), and present a tool supporting our framework (Sect. 10). Finally, we conclude the paper with some directions for future work (Sect. 11).

[2] DataLog plus Vel, i.e., disjunction in Latin.

2 Related work

Modeling requirements is one of the key challenges that secure systems must meet (see Devanbu and Stubblebine's call to arms at ICSE [15]) and a number of researchers have been heeding the call. These proposals can be roughly classified into two main streams.

Object-level modeling uses an off-the-shelves requirement framework, such as UML or i*/Tropos, to model in that framework a number of security requirements. The analysis features of the framework are then used to draw conclusions about the security aspects of the system or to derive guidances for the implementation.

Meta-level modeling takes the same requirements framework, but enhances it with linguistic constructs to capture security requirements. The analysis features and implementation guidances of the framework must then be revised to accommodate the new features.

The advantage of the object-level approach is that reasoning about security is virtually cost-free from the view point of the user: no new language to learn; all (good and bad) features of the modeling framework are immediately usable. If the framework is equipped with a formal semantics and formal reasoning procedures those are also inherited. In the formal framework the "security-notions" are indistinguishable from other requirements. This is also the major disadvantage: the link between security and functional requirements is lost and must be introduced by ad-hoc predicates or relationships by the designer. This makes particularly difficult the modeling of general relationships or rules (such as all processing of personal data should be authorized by the person whose data are being processed).

The meta-level approach trades off readiness for expressiveness. The addition of suitable constructs makes the model more compact and more intuitive to use. This main advantage is coupled by the possibility of designing analysis features that are tailored to the security domain. This is also the key disadvantage: unless the addition of new features is carefully planned, the new framework needs the definition of analysis, semantics, and reasoning procedures. To minimize this problem most sensible approaches try to design the framework in such a way that if one does not use the new features then one can still inherit all of the old framework capabilities.

In early requirements research we can list a number of works at the object-level field. Liu et al. [28] propose a methodological framework for dealing with security and privacy requirements within an agent-oriented modeling framework [42]. They introduce softgoals, as "Security" or "Privacy", to model the corresponding notions, and use dependency analysis to determine the

level of security guaranteed by the system. In [4], general taxonomies for security and privacy are established. These can serve as a general knowledge repository for a knowledge-based goal refinement process. In the field of privacy policy languages a similar approach is taken where notions such as subject, object, and purpose are used. In another paper [29] we show how Hippocratic Database notions [2] can be mapped into Secure Tropos. Finally, Toval et al. [38] present a requirements process model, based upon reuse, together with a reusable template to organize security policies in an organization and a catalog filled with reusable personal data security requirements. In the UML community, Basin et al. [7] present a modeling language, based on UML, called SecureUML. Their approach is focused on modeling access control policies and how these (policies) can be integrated into a model-driven software development process.

On the side of meta-level approaches, a number of works propose to capture security requirements using "standard" software engineering methodologies. Jürjens [24] proposes UMLsec, an extension of the Unified Modeling Language (UML). This proposal uses standard UML extension mechanisms, such as stereotypes, tags, constraints, and profiles, to represent security requirements. For instance, tags {secrecy}, {integrity}, and {authenticity} are used to represent the respective requirements in messages described in UML collaboration and sequence diagrams. McDermott and Fox [33] adapt UML use cases to capture and analyze security requirements. They introduce the notion of abuse case where the result of the interaction is harmful to the system or one of the actors. Sindre and Opdahl [36] define the concept of misuse case, the inverse of a use case, which describes a function that the system should not allow. An analogous proposal has been put forward by van Lamsweerde et al. [40] that introduce the notion of anti-goals, i.e., goals of attackers. A common limitation of these approaches is that they focus on the IT system rather than the organization as a whole and therefore take into account neither trust relations nor the possibility of implementing security requirements with organizational solutions as opposed to technological ones. Moreover, in our previous work [18] we have shown that another key missing concept is the separation of the notions of providing a service and ownership of the very same service.

Indeed, if we look at the requirements refinement process of many proposals, we find out that at a certain stage a leap is made: we have a system with no security features consisting of high-level functionalities, and the next refinement shows encryption, access control, and authentication. The modeling process should instead makes it clear why encryption, access control, and authentication are necessary.

Our own work is well placed within the meta-level modeling field. To avoid some of the disadvantages of the approach we have focused on a modular addition so that dropping all newly proposed features makes us return to the i*/Tropos original methodology. In particular, this work is a step in the direction of closing the gap between the functional and trust requirements and the trust management architecture that is now emerging as the standard way to implement security in distributed systems.

3 Running example

This section describes the high-level functional and security requirements for building a medical IT system that will be used throughout the paper as a running example. Such systems are particularly challenging to model as they have tight and possibly conflicting functional and security requirements. According to current Italian privacy legislation,[3] a *Medical Information System* manages patient information, including information about the medical treatments they have received and provides views of this information to a variety of actors. The system should provide patient medical data if and only if consent is obtained from the patient in question. Patients may refuse to share their personal data if they do not trust the IT system or feel they have not enough control over their data.

In the following sections, we model and analyze such requirements through our methodology in order to determine the correctness and consistency of the entire set of requirements. In particular, we give a practical example of application of our methodology to show its capabilities of supporting system designers in refining and redefining requirements when vulnerabilities and flaws are detected. One particular challenging aspect is showing that the IT system and indeed the organization as a whole fulfills the "need-to-know" principle: private sensitive data should be held only by actors (being them human or software) that actually need it to meet some functional requirements. Every year the law requires organizations to draw and deploy security policies and procedures that guarantee this principle. The

[3] Legge n. 675 del 31 dicembre 1996 (http://www.interlex.it/675/indice.htm). Readers not familiar with Italian may have a look at Ross Anderson's suggestion for privacy policy for the British medical association [3] or have a look at a similar case study we carried out for the University of Trento [31].

ability to model both functional and security requirements is therefore essential.

We start by identifying the main actors (stakeholders):

- *Patient*: the legitimate owner of his personal data, who depends upon the hospital for receiving appropriate health care;
- *Hospital*: the provider of medical treatments, which depends upon the patients in order to have access to their personal information;
- *Clinician*: physician of the hospital that provides specialized medical health advice and, if needed, provides appropriate medical treatment;
- *Health Care Authority* (HCA): the "owner" of medical treatments whose main goals are the fair allocation of resources and the good quality of provided services to the citizens under its authority (opposed to citizens "belonging" to other regional authorities).

From the standpoint of functional requirements, clinicians need fast access to their patients' medical data to provide effective care. Similarly, the HCA needs reliable access to data to allocate effectively available resources and guarantee that each patient can receive good quality medical care. Moreover, the HCA wants to be sure that the system cannot be defrauded in any way and that clinicians and patients act within the scope of their roles.

Further details will unroll in the course of the paper. Certain apparently minor details that our framework can capture (such as who is the "owner" of the goal "provide medical treatments") will later play a major role in identifying the correct authorization and trust management implementation.

4 Secure Tropos

For the embedding of our trust and security primitives we have chosen a state-of-the-art requirements engineering methodology. Among the competing alternatives [33,36,40], we have chosen the i*/Tropos methodology [10], which has been already applied to model security properties [28,41]. One of the main features of Tropos is the role given to the early requirements analysis phase that precedes the prescriptive requirements specification. The main advantage of this phase is that one can capture not only the "what" or the "how", but also the "why" a piece of software is developed. Tropos is also tailored to describe both the organizational environment of a system and the system itself. It uses the concepts of actor, goal, plan, resource, and social dependency. Actors have strategic goals and represent agents (organizational, human or software), roles or positions. A plan (or task) represents a way of fulfilling a goal. A resource represents a physical or an informational entity. Finally, a dependency between two actors indicates that one actor (depender) depends on another (dependee) to accomplish a goal, execute a plan, or deliver a resource. In the remainder of the paper, we generically speak of *providing services* instead of *executing plans*, *fulfilling goals* or *delivering resources* in order to simplify exposition.

Unfortunately, Tropos has been designed with cooperating IT systems in mind. Thus, a dependency between two actors means that the dependee will take responsibility for fulfilling the functional goal of a depender, and the dependee is implicitly authorized to do so. However, it can happen that an actor depends on another for a service, but the dependee is neither the owner of the service nor authorized to provide the service.

Example 1 A patient depends on the hospital for getting medical treatments. On the other hand, the HCA is the legitimate owner of medical treatments. Thus, the hospital must be authorized by the HCA before providing health care.

Furthermore, a Tropos dependency is an intentional relationship: the depender wants a service provided and the dependee is willing and able to provide it. This implicitly assumes that the dependee has the capability to provide the service. However, this assumption is too strong: sometimes, we can depend on another since we know that he knows someone who is willing and able to provide the service (but who we don't directly know).

Example 2 A patient depends on the local hospital for medical treatments. On the other hand, only clinicians are able to provide medical treatments. Thus, the hospital depends on clinicians for providing medical treatments to its patients.

Making explicit who is the legitimate owner of a service, who is entitled to provide a service and who is able to provide a service is the baseline of Secure Tropos.

Intuitively, we have added to Tropos concepts and relationships for representing entitlements, trust, and permission delegation. Further, we have made explicit the concept of capability. Finally, to complete our model we adopt the notion of objectives (i.e., goals) proposed by the Tropos methodology. Table 1 compares the Tropos model with our proposed extension.

Thus, we identify three relations between an actor and a service to express actors' properties and three relations among two actors and a services to express social relationships among actors. In particular, Secure Tropos

Table 1 Comparing tropos model and Secure Tropos model

Tropos Model	**Secure Tropos Model**
• Actor properties 1 Objectives	• Actor properties 1 Objectives 2 Capabilities 3 Entitlements
• Actor relationships 1 Functional dependencies	• Actor relationships 1 Functional dependencies 2 Delegations of permission 3 Trust relations

supports *desires* to represent objectives of actors, *ownership* to represent entitlements of actors, and *provisioning* to represent capabilities of actors. Then, Secure Tropos supports Tropos *functional dependencies*, *delegations of permission*, and *trust relations* for modeling the transfer of entitlements and responsibilities between actors.

The basic idea is that an owner has full authority concerning access and disposition over his entitlements, access to a resource, execution of a plan or fulfillment of a goal. He can also delegate it to other actors. The distinction between owning and providing a service (i.e., a goal, a plan, or a resource) allows us to model situations where, for example, a patient is the legitimate owner of his personal data and the medical IT system provides access to his data. Note that it is possible to "own a goal" in the sense of being entitled to decide who can fulfill it or can provide for its fulfillment.

Example 3 Every hospital can provide medical treatments (eventually delegating to its doctors) but is not authorized to do so. The HCA is the only authorized entity. It delegates permissions to hospitals and also eligibility checks and payment collection.[4]

Notice also the distinction between trust and delegation. Delegation marks a formal passage of authority in the domain of analysis. In contrast, trust capture a social relationship among actors that is not formalized by a "contract". Essentially, a trust relation between two actors indicates the believe of one actor that the other does not misuse some service. In general, by trusting another actor for a service, an actor is sure that the service is properly used.

Example 4 A certificate of entitlement to medical care has to be eventually issued to the patient by the HCA. Such a certificate needs not be digital. Instead, for example, it can be a HCA plastic card with the patient's social security number. With this "certificate", a patient can prove to the hospital that the HCA has delegated to him the permission to get medical treatments.

Example 5 The hospital trusts clinicians to provide medical care only to authorized patients. Such a trust relation represents the belief of the hospital that the clinician will behave in agreement with hospital's policies, and does not represent a "technological" measure to prevent a clinician to provide consultancies to people that have not been previously authorized by the hospital.

Notice the contrast with the HCA–hospital relationships. The HCA trusts the clinicians to fulfill hospital's requirements. Certain hospitals are simply not delegated: without a signed "convenzione" (agreement) their costs will not be reimbursed.

There may be cases, where stakeholders are happy with social protection mechanisms, and other cases where system-supported security mechanisms are needed. Moreover, the distinction between trust and delegation also allows us to model scenarios where some actors must delegate permission on their services to other actors they do not trust. The inconsistencies that triggered by this kind of scenario alert the system designer that the system is not secure and suggest him to introduce mechanisms to ensure actors that their services will not be misused such as monitoring patterns [21]. However, the formal model just offers support to spot inconsistencies, and the decision of what mechanisms to adopt must be taken by the designer.

Now we have all the necessary machinery to start requirements modeling and analysis. Software development in Tropos consists of the following steps:

Early requirements analysis, concerned with the understanding of a problem by studying an existing organizational setting;

[4] This is important in a mixed public/private system such as the Italian one. A person can show up in any clinic (being either public or private in a particular region or in another one) but in absence of an agreement ("convenzione") between the hospital and the HCA certain care facilities though available cannot be used for the individual in question.

Late requirements analysis, where the system-to-be is described within its operational environment, along with relevant functions and qualities;

Architectural design, where the system's global architecture is defined in terms of subsystems, interconnected through data and control flows;

Detailed design, where interactions among actors are specified in terms of inputs, outputs, and control.

In this paper we mainly focus on the requirements analysis phases where various activities contribute to the acquisition of the requirement model and to its refinement into subsequent models.

Actor modeling identifies the actors of the environment and the system's actors and analyzes their goals. During late requirements, actor modeling focuses on the definition of the system-to-be.

Functional dependency modeling identifies actors who depend on one another for obtaining services, and actors which are able to provide services.

Permission trust modeling identifies actors who trust other actors for services, and actors which own services.

Permission delegation modeling consists of identifying actors who delegate to other actors the permission on services.

A graphical representation of the model obtained following these modeling activities is given through three different kinds of *actor diagrams*: *functional model*, *trust model*, and *trust management implementation*. In these diagrams, actors are represented as circles; goals, plans, and resources are respectively represented as ovals, hexagons, and rectangles (Fig. 1). For each social relationships we have a graphical notation to be used by requirements engineers during the design phase. These are shown, along their Tropos counterpart, in Fig. 2.

Notice that for functional dependencies we assume a level of trust among actors for what concerns the ability to provide services. When we say that the hospital functionally depends on the clinician this means that (1) the hospital will delegate the provisioning of the service to the clinician but also (2) it trusts the clinician to be able to do it and take responsibility for it. Here, for the sake of simplicity, we keep the two relations into one. In [21] we have split this relation in its components to explain

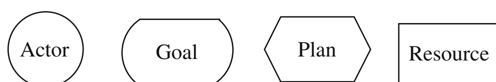

Fig. 1 Graphical representation of Secure Tropos entities

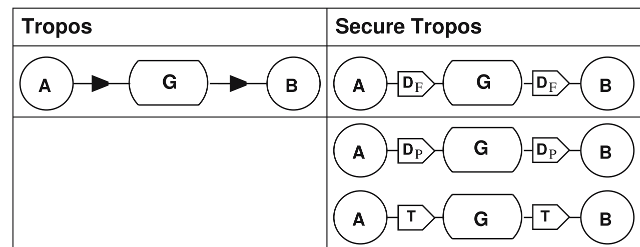

Fig. 2 Graphical notation for actors' relationships

certain monitoring patterns that can be seen in actual systems. Moreover, social relations could be qualified, that is, they are valid only is a certain condition[5] holds.

Once stakeholders have been identified, along with their objectives, entitlements, capabilities, and social relations, the analysis proceeds in order to enrich the model with further details. *Goal modeling* consists of refining requirements through AND/OR-decomposition and eliciting new relations. A graphical representation of goal modeling is given through *goal diagrams* which represent the perspective of specific actors.

5 Modeling the case study

The first activity in the early requirements phase is actor modeling. Analyzing our running example we have identified the following actors. The HCA is an environment actor whose high-level goals are to provide medical treatments to citizens and check equity resource distribution. The Patient is another actor and its goal is having accurate medical treatments. Other actors are the Hospital and the Clinician.

The analysis proceeds introducing social relations between actors. Figures 3, 4, and 5 show, respectively, the trust model, the functional model, and the trust management implementation. In the trust model (Fig. 3), the HCA owns the goal of providing medical treatments, that is, it is entitled to decide who can fulfill this goal and to whom care might be delivered.[6] The HCA trusts the Hospital for providing medical treatments. In turn, the Hospital trusts Clinicians for achieving this goal. However, the Hospital trusts the Clinician to provide medical treatments only to authorized Patients, that is, Patients registered at the local HCA. This explains the trust relation between the HCA and the Hospital for checking equity resource distribution. Finally, Patient owns his

[5] Conditions should be in Horn clausal form in order to easily implement them in our formal framework.

[6] One may argue that everybody should be entitled to have care and not leave the matter to the HCA. Though politically correct this would not correspond to an accurate model of the domain.

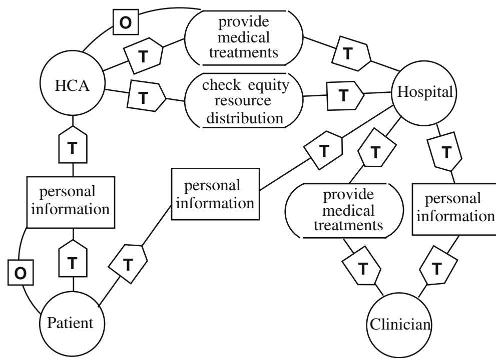

Fig. 3 Early requirements trust model

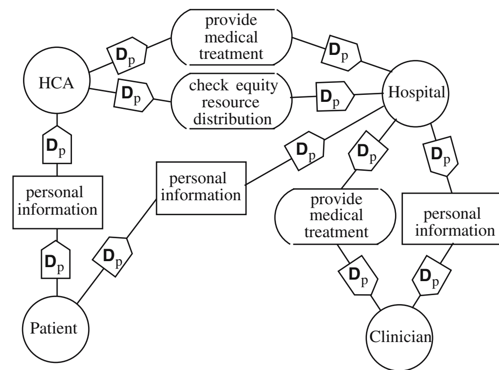

Fig. 5 Early requirements TM implementation

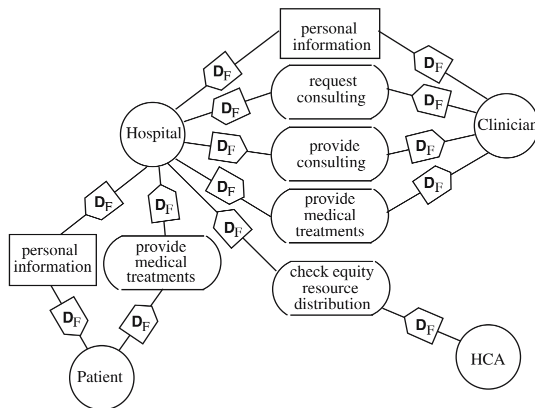

Fig. 4 Early requirements functional model

personal information and trusts that the HCA and Hospital do not misuse it. The Hospital trusts the Clinician for such information.

In the functional model (Fig. 4), the Patient depends on the Hospital for providing medical treatments, and, in turn, the Hospital depends on the Clinician for the fulfillment of this goal. To provide accurate medical care, the Clinician can request specific professional consultancy from the Hospital that, in turn, requests them to other Clinicians. The Clinician also needs patient personal information and requests it from the Hospital. In turn, the Hospital depends on the Patient for getting such information. Also the HCA depends on the Patient for his personal information in order to perform its duties.

In the trust management implementation (Fig. 5), the HCA delegates the permission to check equity resource distribution and to provide medical treatments to the Hospital. In turn, the Hospital delegates the permission to provide medical treatments to the Clinician. Now the Clinician is entitled to provide medical treatments. However, to provide accurate medical care to the Patient, the Clinician need his personal information. Also the HCA needs patient Personal information to perform its duties. On the other side, the Patient signs to the Hospital and HCA the form where he allows the processing of his information. The problem is that the Patient delegates his information to the Hospital, whereas the Clinician does the job, and so there is a mismatch within the kind of delegation. The Patient should delegate his personal information to the Hospital with the possibility of re-delegating it, otherwise things cannot go down to the level where they are needed. If the Hospital got this right, it delegates personal information concerning a specific Patient to the Clinician assigned to that Patient.

An example of goal diagram is presented in Fig. 6. The goal of providing medical treatments is decomposed into access patient record, examination, and prescribe medical treatments. To provide accurate medical care, the Clinician needs patient medical data, but to access them the Clinician must have the consent of their donor. This is modeled by decomposing goal access patient record into get patient consent and get patient information. The analysis reveals the existence of additional social relations between actors, namely a functional dependency

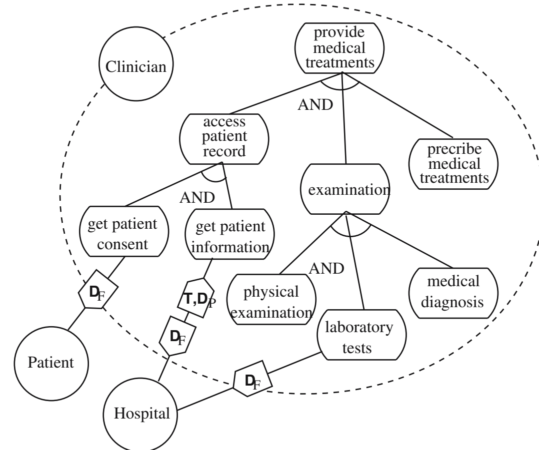

Fig. 6 Goal diagram for the clinician

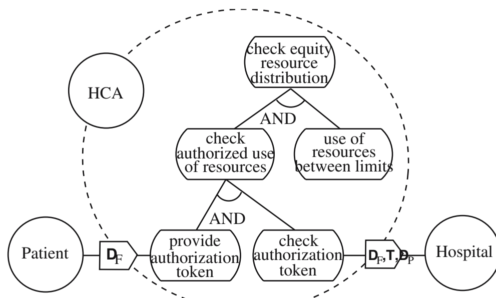

Fig. 7 Goal diagram for the HCA

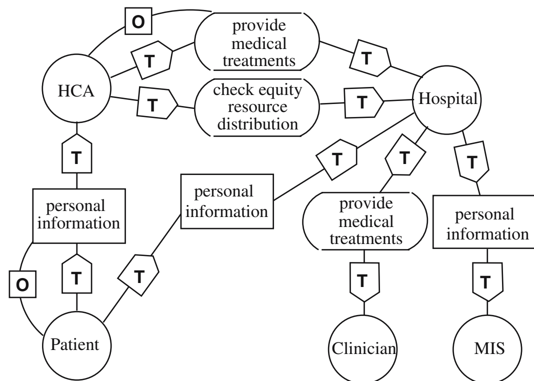

Fig. 8 Late requirements trust model

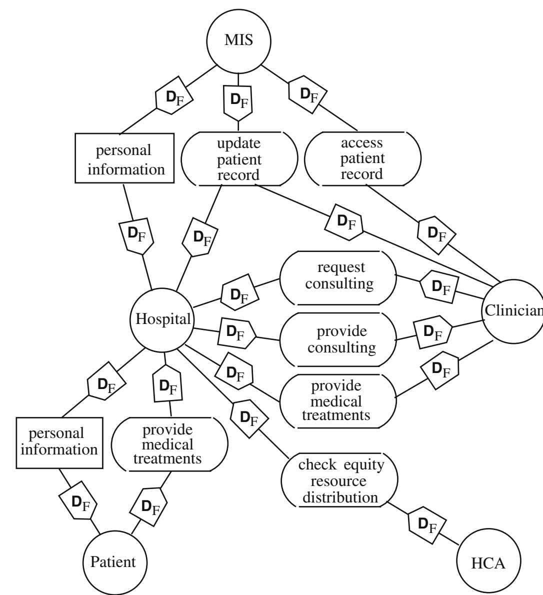

Fig. 9 Late requirements functional model

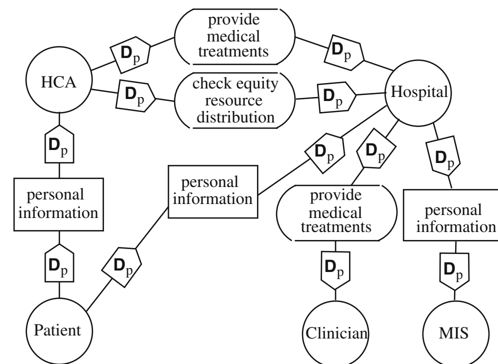

Fig. 10 Late requirements TM implementation

between the Clinician and the Patient for getting patient consent and a functional dependency between the Clinician and the Hospital for getting patient information. The goal examination is decomposed into physical examination, laboratory tests for which the Clinician depends on the Hospital, and medical diagnosis.

Other interesting issues come up by refining goal check equity resource distribution (Fig. 7). This goal is decomposed into check authorized use of resources and use of resources between limits. In turn, check authorized use of resources is decomposed into provide authorization token for which the Patient relies on the HCA, and check authorization token. This refinement reveals how the HCA does not entirely depend on the Hospital for check equity resource distribution, but only for checking authorization token. Thereby, the HCA trusts and delegates the permission to the Hospital only on this subgoal.

For the late requirements analysis phase, we introduce the Medical IT System (MIS) in the model. Often, social relations among actors must be revised upon the introduction of the actor representing the system. Figures 8, 9, and 10 show the trust model, functional model, and trust management implementation, respectively. The analysis reveals that the Clinician does not directly depend on the Hospital for patient personal information since the Hospital store information into the MIS and actors (such as Clinician) who need such information for performing their duties, directly retrieve it from the MIS. Actually, the main objectives of the MIS are to provide access to patient records and to maintain updated patient records. For these goals, the MIS depends on the Clinician and on the Hospital. In particular, the MIS depends on the Clinician for medical data and on the Hospital for vital statistics data.

Figure 11 shows the goal refinement of access patient record. This goal is decomposed into authenticate clinician, get patient consent, and patient record. The analysis reveals that the Clinician relies on the MIS to get patient consent. Once the Clinician has been authenticated, the MIS requires the patient consent to the Hospital before disclosing the record. Then, when above goals are achieved, the MIS delegates the permission on the patient record to the Clinician who required such a record.

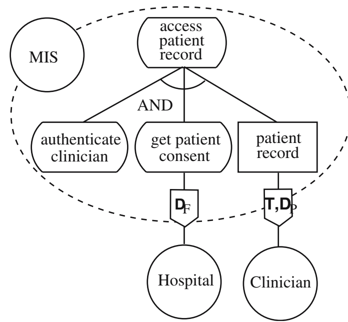

Fig. 11 Goal diagram for the medical IT system

6 Formalization

To build the formal semantics of the framework, we use Datalog [1], following the example of others who have worked on access control and trust management [26,27].

Datalog is a language of facts and rules. The facts are called *extensional predicates*. In contrast, the relations computed by applying rules are called *intensional predicates*. In other words, extensional predicates appear only in the body of clauses or as ground facts. Intensional predicates are the only ones that can appear both in the head and in the body of clauses. Intuitively, extensional predicates correspond to the drawings of the requirements engineer, while intensional predicates are derived by the system.

Both extensional and intensional predicates used to model the concepts and relations of ECO model are listed in Table 2. The first batch of predicates are used to model the functional model. When an actor has the capabilities to fulfill a service, he provides it. The intuition is that provides(a, s) holds if instance a provides the corresponding instance of s. The predicates requests(a, s) and aims(a, s) hold if actor a has the objective of getting s. The first is the extensional predicate and the latter is the intensional predicate. The relation depends(a, b, s) holds if actor a depends on actor b for the fulfillment of service s. The predicate fulfills(a, s) is true when s service is provided by actor a, and satisfies(a, s) holds if actor a is sure that someone delivers service s. Using both the last two predicates could seem redundant, but the first identifies the actor who really provides the service and is used as a start point of the chain used to identify the actors confident that the service will be provided.

Table 2 Predicates

Actor Primitive Properties
provides(Actor: a, Service: s) requests(Actor: a, Service: s) owns(Actor: a, Service: s)
Actor Derived Properties
has_per(Actor: a, Service: s, $\mathcal{N}^+ \cup \{*\}$: n) aims(Actor: a, Service: s) fulfills(Actor: a, Service: s) satisfies(Actor: a, Service: s)
Actor Primitive Relationships
depends(Actor: a, Actor: b, Service: s) trust($perm$, Actor: a, Actor: b, Service: s, $\mathcal{N}^+ \cup \{*\}$: n) delegate($perm$, Actor: a, Actor: b, Service: s, $\mathcal{N}^+ \cup \{*\}$: n)
Actor Derived Relationships
trustChain($perm$, Actor: a, Actor: b, Service: s, $\mathcal{N}^+ \cup \{*\}$: n)

Next, we have predicates for the trust model. The extensional predicate owns*(a,s)* holds if actor a owns service s. The predicate trust$(perm, a, b, s, n)$ holds if actor a trusts actor b for service s; n is called *trust depth*. As suggested by Li et al. [26] for their Delegation Logics, trust has depth which is either a positive integer or "*" for unbounded depth. Trust depth represents how much an actor takes into account the judgment of another actor. In particular, depth equal to 1 means that an actor trusts another actor, but he does not trust the trustee's judgment; depth equal to 2 means that an actor trusts another actor, and all actors the trustee trusts directly, and so on. Finally, unbounded trust means that an actor trusts another unconditionally. In practice a depth of 1 or 2 seems sufficient. Unbounded but qualified depth is also useful: we trust our general practitioner, and all doctors recommended by him and the doctors recommended by those and so on. Thus, we have unbounded trust relative to a particular goal: providing care by registered physicians. However, we do *not* allow for blanket trust, not even in the language itself. As in delegation logic [26], trust is always linked to a particular goal, plan, or resource. This is also the case for RT_2 but not for other languages in the RT family [27]. The predicate trustChain is essentially the intensional version of trust and is used to model trust chains between actors. Actually, trustChain literals take the same arguments of trust, and are computed by applying rules (see Appendix A).

Every trust management framework is based on credentials and their delegation, so the extensional predicate delegate$(perm, a, b, s, n)$ holds if actor a delegates the permission to use the service s to actor b; n is called *delegation depth*. As trust relations, delegations have depth. One way to view depth is the number of re-delegation steps that are allowed; depth 1 means that no re-delegation is allowed, depth n means that $n - 1$ further step are allowed, and depth "*" means that unbounded re-delegation is allowed. Using delegation and trust depth without goal or services would be an unsatisfactory practical approximation. Indeed, most actual delegation or trust relationships are "qualified". An example is the trust in the highly reputed head of a clinical department and direct aides. Another example is delegation on authorization to the head of clinical department or in his absence to the most senior aide on

duty. The basic idea of has_per is that whoever has a service, has authority concerning access and disposition of the service, and he can also delegate this authority to other actors. Also has_per has depth which is either a positive integer or "*" for unbounded depth. The intuition is that we are eventually entitled to pass our entitlements to others. This is fairly standard in contractual relations for a small depth. They can be easily accommodated simply by decomposing the goal into subgoals, i.e., providing care can be refined by providing care by the head of the medical department or providing care by a member of the department. We want to remark that delegate is put by designer, while has_per is computed by the system. In summary, has_per represents implicitly chains of delegation of permission starting from the owner of the service. We refer to Appendix A for the list of rules used to derive has_per literals.

A critical phase of the system development process is the analysis of requirements in order to detect inconsistencies in the requirements. In order to cope with this issue, we adopt the following verification process:

– model the system by using Secure Tropos primitives;
– complete the extensional description of the system;
– verify the correctness and consistency of system requirements.

The first step of requirements analysis process is the modeling phase. This phase addresses to identify actors along their objectives, capabilities and entitlements, and the social relations used to transfer them to other actors. The application of the Secure Tropos methodology to complex case studies [31,32] has shown how the introduced primitives are right ones to model most of the system requirements. Thus, this phase requires system designers to represent system requirements (expressed in the form of actors' properties and relations) by using extensional predicates.

Once the requirements engineer has drawn up the model (i.e., the extensional predicates), he may want to verify the correctness and consistency of functional, trust, and security requirements. Unfortunately, the extensional description of the system cannot be used to perform an accurate analysis. Thus, for getting the right conclusions from an intuitive model, we need to complete the model using rules that we call *axioms*. Essentially, axioms define the semantics of the framework and are used to make explicit that information needed for requirements verification. Table 3 provides a description of the axioms supported by Secure Tropos, while their formalization is presented in Appendix A.

Once the comprehensive description of the system (i.e., the intensional predicates) is derived, system designers are able to verify if the model complies with some desirable *properties*. Essentially, they want to check the consistency of the model to guarantee that functional, trust, and security requirements are not self-contradictory. Thus, we have provided a set of properties representing the compliance of the system with authorization and availability requirements. If all the properties are not simultaneously satisfied, the system is inconsistent. In other words, if some properties are not verified, the system is not secure. Table 4 describes the properties supported by Secure Tropos, while their formalization is presented in Appendix A.

Table 3 Axioms

Functional Requirement Model
This batch of axioms determines actors' responsibilities according to the functional dependencies drawn by the system designer.
Trust Requirement Model
This batch of axioms builds the trust network, that is, the full set of trust chains among actors.
Trust Management Implementation
This batch of axioms identifies actors' entitlements, actors who fulfill services, and actors confident that their objectives will be satisfied.
Goal Refinement
This batch of axioms propagates actors' objectives, entitlements and capabilities through goal refinement.

Table 4 Properties

Availability Requirements
This batch of properties verifies if the current model is such that every actor can satisfy his objectives (e.g., functional dependency ends in actors with the appropriate capabilities and rights).
Authorization Requirements
This batch of properties verifies if actors who delegate the permission on a service have enough rights for this, if delegators trust delegatees, if only trusted actors can access and fulfill a service, and if actors who fulfill a service are entitled to do so.

We want to remark that properties are different from axioms (that must be true for every model) and so the designer may well be perfectly happy with a design, say, that satisfies only a part of the properties we have defined. There may be applications where this is acceptable because alternatives are too costly. In some cases the failure of properties demands the presence of additional security mechanisms to guarantee security protection. For instance, if an actor must delegate the permission on a service to an untrusted actor, designers could introduce some mechanisms, such as a monitor [21], to check that the delegatee will not misuse the service.

7 Formalizing the case study

The Secure Tropos formal framework has been implemented in the DLV system [25] in order to allow an

automatic verification of the requirements consistency. Figures 19 and 20 (Appendix B) present, respectively, the axioms and properties of our formal framework in the form of DLV statements. In the remainder of this section, we show an application of the requirements verification process assisted by the DLV system and, in particular, how the analysis aids system designers in refining system requirements.

A basic property system designers want to verify is whether only the clinicians assigned to a certain patient are trusted to access information about that patient. This property can be easily implemented into the DLV system with the following constraint.

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

Essentially, this constraint checks if there is a trust chain (with arbitrary trust depth) from a patient to a clinician for a patient's record where the clinician has not been assigned by the hospital to that patient.

We now analyze the trust model concerning patient personal information presented in Fig. 8. Below we list the trust relations between actors, and present their formalization in Fig. 12.

1. The Patient trusts completely the HCA;
2. The HCA trusts completely the Hospital;
3. The Hospital trusts completely the Medical IT System;
4. The Medical IT System trusts completely the Clinician.

To verify the correctness and consistency of the organizational model, system designers may instantiate the model by introducing occurrences of actors. We suppose that in the system there are ten occurrences of patient and three of clinician. Since instances inherit the relations associated with their meta classes [22], the DLV system completes the model by computing thirty trust chains, one for each patient/clinician instance pair. Thus, when the DLV system is used to verify the above property, it reports an inconsistency: actually, all clinicians are authorized to access personal information of every patient. Ideally, we would authorize only the clinicians assigned to a patient to access his data. Moreover, the analysis reveals that clinicians are authorized to re-delegate patient information. These inconsistencies can be removed by restricting the trust relation between the Medical IT System and the clinician as follows

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

The refined result guarantees that patient consent must be sought for any other agent, such as clinician colleagues, to be able to access patient information, and the patient must be notified of every access. This complies with the requirements drawn in Sect. 5 where a clinician has to request a consultation with colleagues through the hospital (Fig. 4) and the patient must give the permission to access the data. Applying the verification process to the new model, the DLV system does not reveal any inconsistencies.

System designers might erroneously introduce some occurrences that do not behave correctly. For instance, they can introduce an occurrence of clinician that directly requires a consultation to a colleague (i.e., another occurrence of clinician) and so delegate patient information to him without the permission of the hospital. In this case the DLV system spots inconsistencies since properties are not verified. This proves the correctness of the model because it is able to detect incorrect behaviors.

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. [28] in their security requirements model based on Alloy. Least privilege [35] requires that actors should be assigned only the permissions needed to perform their assigned duties and functions.

Example 6 To check if a clinician has only medical records of his patients, we use the following property:

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

```
trust(perm,Pat,X,Rec,*):- isHCA(X),owns(Pat,Rec).
trust(perm,hca,hospital,Rec,*):- isRecord(Rec).
trust(perm,hospital,mIS,Rec,*):- isRecord(Rec).
trust(perm,mIS,X,Rec,*):- isClinician(X),
                          isRecord(Rec).
```

Fig. 12 Trust relations in DLV system

8 Negative authorizations

In all practical examples of policies and requirements for e-health, we found the need for negative authorizations (for non-functional requirements) and negative goals or goals whose fulfillment restrains the fulfillment of other goals (for functional requirements). Tropos already accommodates the notion of positive or

Table 5 Negative Authorization Predicates

Primitive negative authorization
delDenial(Actor: a, Actor: b, Service: s, $\mathcal{N}^+ \cup \{*\}$: n)
prohibition(Actor: a, Actor: b, Service: s)
Derived negative authorization
delDChain(Actor: a, Actor: b, Service: s, $\mathcal{N}^+ \cup \{*\}$: n)
prohibitionChain(Actor: a, Actor: b, Service: s)

negative contribution of goals to the fulfillment of other goals. We only need to lift the framework to delegation and trust. Notice that having negative authorization in the requirements model does *not* mean that we must use "negative" certificates. Even if some form of negative certificates is used in real life,[7] we use negative authorizations to help designers in shaping the perimeter of positive trust, i.e., positive certificates, to avoid incautious delegation certificates that may give more powers than desired.

The framework presented in previous sections is based on a *closed world* assumption, for which the lack of authorization is considered as a negative authorization. Essentially, whenever an actor tries to access an object, if a positive authorization is not found in the system, the actor is denied the access. This approach has a major problem: the lack of an authorization for a given actor does not prevent such an actor from receiving the authorization from another actor.

Suppose that an actor should not be given access to a service. In decentralized authorization administration, an actor possessing the right to use the service can delegate the authorization on that service to the wrong actor. Since many actors may have the right to use the service, it is not always possible to enforce with certainty the constraint that an actor cannot access such a service. Thus, we propose an explicit *negative authorization* as an approach for handling this type of constraint.

An explicit negative authorization expresses a *denial* for an actor to access a service. In our approach negative authorizations take precedence on positive ones. That is, whenever a user has both a positive and a negative authorization on the same object, the user is prevented from accessing the object. Essentially, negative authorizations in our model are handled as blocking authorizations: whenever an actor receives a negative authorization, his positive authorization becomes blocked.

We distinguish two predicates (Table 5) for the extensional features put down by the designer: delDenial and prohibition. The intuition is that delDenial(a, b, s, n) holds if actor a delegates the permission to denial the service s to actor b, and prohibition(a, b, s) holds if actor a forbids to use the service s to actor b, that is, actor a says that service s cannot be assigned to actor b. We assume that if an actor a denial an actor b to have service s there is not a delegation chain from a to b. Thus, if a is the owner of s then b cannot have s. Otherwise, b could have s if there exists a delegation chain from owner of s and b without a.

As done for positive authorizations, we provide the corresponding intensional predicates delDChain and prohibitionChain. These predicates are derived by the system and are used to build denial chains and prohibition chains. Fig. 17 (Appendix A) presents the axioms to build such chains while their translation into DLV specifications is given in Fig. 21 (Appendix B).

Our framework supports automatic verification of security requirements also in the presence of negative authorization. Fig. 18 (Appendix A) presents this set of properties. Next, we present an example of such properties.

Example 7 A basic property we want to check is "if actor A who owns service S prohibits actor B, then B cannot have S (from other intermediate actors)" (this statement corresponds to P7 in Fig. 18). This can be represented in the DLV system by the following constraint:

```
:- prohibitionChain(A,B,S), owns(A,S),
   has_per(B,S,N).
```

Suppose that a patient requires a new clinician, since he distrusts the old one. If the Medical IT System did not update this information, the last can give the patient data to the old clinician. Applying the above constraint to this scenario, the DLV system reports an inconsistency: an unauthorized actor can access private data.

9 RT implementation

This section presents the implementation of our approach in the RT framework [27]. See [8,26] for alternative systems. Our choice has been especially motivated by the semantics based on logic so that translation between ours and the RT framework is well founded.

RT entities are individuals that can issue credentials and make requests. RT represents roles as attributes. An entity plays a role if and only if the entity is entitled to have the attribute representing the role. A role has the form $A.R$ where A is an entity and R is a role name. RT allows an entity to assert that another entity has a certain attribute by using credentials. In particular, RT supports four kinds of credentials. Each of them refers to a way

[7] E.g., a certificate issued by the government that you have no pending criminal trials or a court decision that bans you from public office.

```
1  for each owns(A, S) do
2      begin
3          A.S ⟵ A;
4          A.S ⟵ A.S;
5      end
6  for each delegate(perm, A, B, S, N) do
7      begin
8          A.S ⟵ B;
9          if N > 1 then A.S ⟵ B.S;
10     end
```

Fig. 13 From Secure Tropos to RT

to define which entities are entitled to play a role, that is, to have an attribute. The first type of credential is a simple assignment, and has the form $A.R \longleftarrow B$. This credential means that entity A says that entity B has the attribute R. The second takes the form $A.R \longleftarrow B.R_1$ and means that A authorizes all the entities authorized by B to play the role R_1, to play also the role R. The third type has the form $A.R \longleftarrow A.R_1.R_2$ where $A.R_1.R_2$ is called a *linked role*. This credential means that A gives the permission to play its role R to all entities playing the role $B.R_2$ in which B is playing the role $A.R_1$. The last type has the form $A.R \longleftarrow f_1 \cap f_2 \cap \cdots \cap f_k$ where $f_1, \ldots, f_k$ can be an entity, a role, or a linked role starting with A, and $f_1 \cap f_2 \cap \cdots \cap f_k$ is called *intersection*. This kind of credential represents a conjunction of credentials.

The RT framework has a logic-based semantic foundation based on Datalog and this is essentially defined through a function from roles to sets of entities. We base our translation on this semantics.

Figure 13 presents an algorithm to map our framework into RT. Lines 1–5 deal with the predicate owns, and lines 6–10 with the predicate delegate. In particular, line 3 defines that the owner of service S is entitled to provide service S. This corresponds to Ax6 of our framework. Line 4 represents the owner's right to delegate the permission on his service to other actors. In the second part of the algorithm, line 8 states that the delegatee is entitled to provide service S, and line 9 that if the delegation depth is grater than 1, the delegatee is entitled to re-delegate the permission on services S.

Example 8 A patient allows his clinician to read his medical data in order to receive accurate medical care. We express this policy in our framework as

$$\mathsf{delegate}(perm, Pat, Cli, Rec, 1) \leftarrow \mathsf{isClinicianOf}(Pat, Cli) \wedge \mathsf{owns}(Pat, Rec)$$

The intuition is that $\mathsf{isClinicianOf}(a, b)$ holds if instance a is the clinician of instance b. The above statement is translated in the RT framework as

$$\mathsf{Pat.Rec} \longleftarrow \mathsf{Pat.clinician}$$

If the fact $\mathsf{owns}(Pat, Rec)$ occurs in the extensional description of the system, we find the following credentials in the RT framework

$$\mathsf{Pat.Rec} \longleftarrow \mathsf{Pat}$$
$$\mathsf{Pat.Rec} \longleftarrow \mathsf{Pat.Rec}$$

These statements correspond, respectively, to openness privacy principle [2] for which the data owner can access all information about him stored into the database, and to the right of the owner to delegate permission to read his own information to other actors. Given the statement "Pat.clinician ⟵ Cli" stating that Cli is the clinician of patient Pat, one can conclude that "Pat.Rec ⟵ Cli", that is, Cli is entitled to access record Rec.

10 ST-Tool

Our framework with all features described in this paper is supported by the ST-Tool.[8] Basically, the ST-Tool is a CASE tool in which it is possible to draw and verify Secure Tropos models. The screen (Fig. 14) is divided in four main areas: the menu of Secures Tropos elements on the top, the graphical editor in the middle, the menu for choosing different representations of models at the bottom, and the properties viewer and editor at the left. Essentially, the ST-Tool allows system designers to draw Secure Tropos diagrams by selecting from the top menu the desired Secure Tropos elements and to edit and verify its properties in the left menu.

Moreover, the tool allows an automatic transformation from Secure Tropos graphical models into formal specifications. In particular, the tool supports the transformation into Datalog specification. The resulting specification is automatically displayed by selecting the corresponding panel in the down menu. This specification then is used for an automatic verification of models. To this end, ST-Tool provides a user friendly interface within the DLV system and other datalog-based solvers, namely ASSAT,[9] Smodels,[10] Cmodels.[11] Actually, the tool permits system designers to select the security properties they want to verify and to complete models with additional "ad-hoc" Datalog statements related to the specific domain. Once designers are confident with the model, the tool passes the entire set of specifications (i.e., the extensional description of the model, the axioms, the selected properties, and the additional Datalog statements) to external solvers. Once

[8] http://sesa.dit.unitn.it/sttool/

[9] http://assat.cs.ust.hk/

[10] http://www.tcs.hut.fi/Software/smodels

[11] http://www.cs.utexas.edu/users/tag/cmodels.html

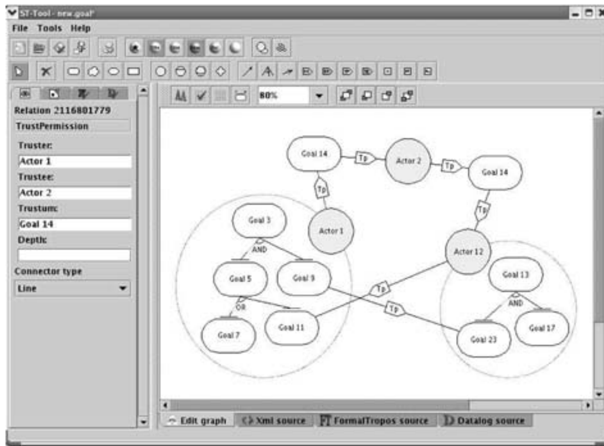

Fig. 14 ST-Tool

the selected solver completes the analysis, the output is parsed and displayed in a user-readable format by the tool.

11 Conclusion

The main contribution of this paper is to bridge the gap between functional and trust requirements of an IT system and its TM/PMI architecture. In particular, we have proposed a requirements specification and an analysis framework based on the clear separation of trust and delegation relationships. This distinction makes it possible to capture high-level security requirements without being immediately bogged down into considerations about cryptographic algorithms or security implementation. This is analogous to what is supposed to happen when discussing functional requirements: one does not get immediately trapped into discussions about implementation details, Java patterns, and coding techniques.

Further, the framework supports the automatic verification of system requirements specified in a formal modeling language. Our formal framework is based on Delegation Logics [26] and RT [27]. However, there are some differences from these frameworks. The main difference is that we distinguish trust from delegation, where trust can be seen as a mental state based on beliefs [11] and delegation as the effective transfer of rights. Our framework differs also from RT because we consider delegation and trust depth, while RT support only unbounded re-delegation. A reason for limiting delegation with a depth control is that trust is not a transitive relationship. Then, we have used the DLV system to check system consistency. Finally, we have defined the trust management implementation of our framework into the RT framework.

The research presented here is still in progress. Much remains to be done to further refine the proposed framework. However, its usefulness has already been tested with real case studies [31,32]. We are currently working in the direction of incorporating explicitly roles and adding time features. Actually, Secure Tropos supports only a static notion of trust. The introduction of time features will allow us to support dynamic aspect of the system, as a dynamic notion of trust. We are also investigating the effects of supporting hierarchies of objects and hierarchies of actors.

Acknowledgements This work was partly supported by the projects RBNE0195K5 FIRB-ASTRO, 016004 IST-FP6-FET-IP-SENSORIA, 27587 IST-FP6-IP-SERENITY, 2003-S116-00018 PAT-MOSTRO, 1710SR-B/P PAT-STAMPS.

Appendix A: Formal framework

In order to complete and verify the model, we use Datalog [1]. A Datalog program is a set of rules of the form $L \leftarrow L_1 \wedge \cdots \wedge L_n$ where L, called head, is a positive literal and $L_1, \ldots, L_n$ are literals and they are called body. Intuitively, if $L_1, \ldots, L_n$ are true in the model then L must be true in the model. In Datalog, negation is treated as negation as failure. In other words, if there is no evidence that an atom is true, it is considered to be false. We refer to [1] for a discussion about complexity, soundness and completeness issues concerning the use of Datalog. Next, we present the axioms and properties supported by our formal framework.

A.1 Axioms

Axioms define the semantics of our framework and are used to complete the extensional description of the system. In particular, our formal framework supports all phases of the requirements analysis process described in the paper, including goal modeling. Thus, firstly, we introduce predicates for goal/task refinement and resource decomposition (Table 6). Predicate service(s) holds if s is a service. Predicate subservice(s_1, s) holds if s_1 is a subservice of s. Predicate OR_subservice(s_1, s) holds if s_1 is derived from s through OR-decomposition, and AND_subservice(s_1, s) holds if s_1 is derived from s through AND-decomposition. As we mentioned,

Table 6 Goal refinement predicates

Goal refinement
service(Service: s)
subservice(Service: s_1, Service: s_2)
OR_subservice(Service: s_1, Service: s_2)
AND_subservice(Service: s_1, Service: s_2)

S1 $\mathsf{subservice}(S_1, S) \leftarrow \mathsf{OR_subservice}(S_1, S)$
S2 $\mathsf{subservice}(S_1, S) \leftarrow \mathsf{AND_subservice}(S_1, S)$
S3 $\mathsf{subservice}(S, S) \leftarrow \mathsf{service}(S)$
S4 $\mathsf{subservice}(S_2, S) \leftarrow \mathsf{subservice}(S_2, S_1) \wedge \mathsf{subservice}(S_1, S)$
S5 $\mathsf{service}(S_1) \leftarrow \mathsf{subservice}(S_1, S)$

Fig. 15 Axioms for subservice

Functional Requirement Model

Ax1: $\mathsf{aims}(A, S) \leftarrow \mathsf{requests}(A, S)$
Ax2: $\mathsf{aims}(B, S) \leftarrow \mathsf{depends}(A, B, S) \wedge \mathsf{aims}(A, S)$

Trust Requirement Model

Ax3: $\mathsf{trustChain}(perm, A, B, S, N) \leftarrow \mathsf{trust}(perm, A, B, S, N)$
Ax4: $\mathsf{trustChain}(perm, A, C, S, P) \leftarrow \mathsf{trustChain}(perm, A, B, S, N) \wedge \mathsf{trustChain}(perm, B, C, S, M) \wedge P = \min\{N-1, M\} \wedge N \geq 2$
Ax5: $\mathsf{trustChain}(perm, A, B, S, N-1) \leftarrow \mathsf{trustChain}(perm, A, B, S, N) \wedge N \geq 2$

Trust Management Implementation

Ax6: $\mathsf{has_per}(A, S, *) \leftarrow \mathsf{owns}(A, S)$
Ax7: $\mathsf{has_per}(A, S, N-1) \leftarrow \mathsf{has_per}(A, S, N)$
Ax8: $\mathsf{has_per}(B, S, P) \leftarrow \mathsf{delegate}(perm, A, B, S, M) \wedge \mathsf{has_per}(A, S, N) \wedge P = \min\{N-1, M\} \wedge N \geq 2$
Ax9: $\mathsf{fulfills}(A, S) \leftarrow \mathsf{has_per}(A, S, N) \wedge \mathsf{aims}(A, S) \wedge \mathsf{provides}(A, S)$
Ax10: $\mathsf{satisfies}(A, S) \leftarrow \mathsf{fulfills}(A, S)$
Ax11: $\mathsf{satisfies}(A, S) \leftarrow \mathsf{depends}(A, B, S) \wedge \mathsf{satisfies}(B, S)$

Goal Refinement

Ax12: $\mathsf{has_per}(A, S', N) \leftarrow \mathsf{has_per}(A, S, N) \wedge \mathsf{subservice}(S', S)$
Ax13: $\mathsf{fulfills}(A, S) \leftarrow \forall S_i\ \mathsf{AND_subservice}(S_i, S) \wedge \mathsf{fulfills}(A, S_i)$
Ax14: $\mathsf{fulfills}(A, S) \leftarrow \mathsf{fulfills}(A, S') \wedge \mathsf{OR_subservice}(S', S)$
Ax15: $\mathsf{satisfies}(A, S) \leftarrow \forall S_i\ \mathsf{AND_subservice}(S_i, S) \wedge \mathsf{satisfies}(A, S_i)$
Ax16: $\mathsf{satisfies}(A, S) \leftarrow \mathsf{satisfies}(A, S') \wedge \mathsf{OR_subservice}(S', S)$

Fig. 16 Axioms

services are actually refined in the classification of goal, plan, and resource. Thus, in the actual language used for writing requirements we have the notion of sub-plan, the relation part-of for resources and so on. Figure 15 presents the axioms for subservices. Their meaning is immediate.

Figure 16 presents the axioms specific to our framework. Ax1-2 say that an actor has as objective to fulfill a service if either this is his own objective or the objective of another actor that depends on (i.e., has functionally delegated to) him the fulfillment of such an objective. Ax3-4 are used to complete the trust network among actors. Ax5 propagates trust relations through depth. Essentially, if someone trusts with certain depth, then he trusts with smaller depth. Ax6 states that actors have full authority on his own services, and Ax7 propagates permission through depth. Essentially, if someone is entitled with certain depth, then he is also entitled with smaller depth. Ax 8 says that the delegatee is entitled to provide the service. Ax9 states that actors who have permission on a service that belongs to their objectives and also have the capability to provide it, can fulfill the service. Ax10 states that actors that fulfill a service are confident that the service is satisfied, and Ax11 propagates this confidence along dependency relations. Ax12-16 treat the cases of goal refinements and how

Trust Management Implementation

Ax8: $\mathsf{has_per}(B, S, P) \leftarrow \mathsf{delegate}(perm, A, B, S, M) \wedge \mathsf{has_per}(A, S, N) \wedge \text{not } \mathsf{prohibitionChain}(A, B, S) \wedge P = \min\{N-1, M\} \wedge N \geq 2$
Ax17: $\mathsf{delDChain}(A, B, S, N) \leftarrow \mathsf{delDenial}(A, B, S, N)$
Ax18: $\mathsf{delDChain}(A, C, S, P) \leftarrow \mathsf{delDChain}(A, B, S, N) \wedge \mathsf{delDChain}(B, C, S, M) \wedge P = \min\{N-1, M\} \wedge N \geq 2$
Ax19: $\mathsf{prohibitionChain}(A, B, S) \leftarrow \mathsf{prohibition}(A, B, S)$
Ax20: $\mathsf{prohibitionChain}(A, C, S) \leftarrow \mathsf{delDChain}(A, B, S, N) \wedge \mathsf{prohibition}(A, C, S) \wedge N \geq 1$

Fig. 17 Negative authorization axioms

the various predicates are differently re-evaluated after refinements.

In order to accommodate negative authorizations in the formal framework some axioms should be added and others modified. Figure 17 shows the list of such axioms. Ax8 is modified to support the "negative authorizations take precedence" policy. Ax17-18 are used to build denial chains and Ax19-20 to build prohibition chains.

A.2 Properties

Once the intensional description of the system is derived, designers may want to verify if the system complies with trust and security requirements. Thus, we provide a set of *properties* representing high-level security requirements that system designers can include in the model. Properties are different from axioms: they are desirable design features, but may not be true of the particular design at hand. In Fig. 18 we use $A \Rightarrow ?\ B$ to mean that each time A holds it is desirable that B also holds. In Datalog this can be represented as the constraint :- A, *not* B.

Figure 18 presents the properties used to verify the correctness and consistency of security requirements. P1 implements availability requirements. It checks if an actor is confident that his objectives will be satisfied.

Other properties focus on the verification of authorization requirements. P2 verifies if an agent that delegates a service is entitled to do it. Essentially, actors must have enough rights to delegate the service. P3 verifies if the delegator trusts the delegatee: actors can only delegate to agents that they trust. Rights or privileges can

Availability Requirements

P1: $\mathsf{requests}(A, S) \Rightarrow ?\ \mathsf{satisfies}(A, S)$

Authorization Requirements

P2: $\mathsf{delegate}(perm, A, B, S, N) \Rightarrow ?\ \exists M \geq N\ \mathsf{has_per}(A, S, M)$
P3: $\mathsf{delegate}(perm, A, B, S, N) \Rightarrow ?\ \exists M \geq N\ \mathsf{trustChain}(perm, A, B, S, M)$
P4: $\mathsf{has_per}(B, S, N) \wedge \mathsf{owns}(A, S) \wedge A \neq B \Rightarrow ?\ \mathsf{trustChain}(perm, A, B, S, N)$
P5: $\mathsf{fulfills}(B, S) \wedge \mathsf{owns}(A, S) \wedge A \neq B \Rightarrow ?\ \exists N\ \mathsf{trustChain}(perm, A, B, S, N)$
P6: $\mathsf{fulfills}(A, S) \Rightarrow ?\ \exists N\ \mathsf{has_per}(A, S, N)$

Authorization Requirements with Negative Authorizations

P7: $\mathsf{prohibitionChain}(A, B, S) \wedge \mathsf{owns}(A, S) \Rightarrow ?\ \text{not } \mathsf{has_per}(B, S, N)$
P8: $\mathsf{prohibitionChain}(A, B, S) \wedge \mathsf{owns}(A, S) \Rightarrow ?\ \text{not } \mathsf{fulfills}(B, S)$

Fig. 18 Desirable properties of a design

be given to trusted actors that are then responsible for actors they may delegate this right to. This forms a delegation chain. If any actor along this chain fails to meet the requirements associated with a delegated right, the chain is broken and all actors following the failure are not permitted to perform the action associated with the right. P4 states that only actors authorized by the service owner has authority concerning access an disposition of the service. P5 states that if an actor fulfills a service owned by another actor, he must be trusted by the legitimate owner. This means that only actors authorized by the service owner can fulfill the service. P6 checks whether actors that provide a service are entitled to do this.

The last batch presents properties for verifying security requirements in presence of negative authorizations. P7-8 verify that, if the owner of a service forbids to use it to another actor, the latter cannot have and fulfill the service.

Appendix B: Mapping the formal framework into DLV system

Figures 19 and 20 present, respectively, the axioms presented in Fig. 16 and the properties in Fig. 18 in the corresponding DLV statements.

```
aims(A,S):- requests(A,S).
aims(B,S):- depends(A,B,S), aims(A,S).

trustChain(perm,A,B,S,N):- trust(perm,A,B,S,N).
trustChain(perm,A,B,S,N):-#succ(N,M),trustChain(perm,A,B,S,M),
                          N>0.
trustChain(perm,A,C,S,P):- #succ(P,N), trustChain(perm,A,B,S,N),
                           trustChain(perm,B,C,S,M), M>=N, N>1.
trustChain(perm,A,C,S,M):- trustChain(perm,A,B,S,N),
                           trustChain(perm,B,C,S,M), N>M, N>1.

has_per(A,S,*):- owns(A,S).
has_per(A,S,N):- #succ(N,M), has_per(A,S,M), N>0.
has_per(B,S,P):-#succ(P,N),delegate(perm,A,B,S,N),has_per(A,S,M),
                M>=N, N>1.
has_per(B,S,M):- delegate(perm,A,B,S,N), has_per(A,S,M), N>M, N>1.
fulfills(A,S):- has_per(A,S,N), aims(A,S), provides(A,S).
satisfies(A,S):- fulfills(A,S).
satisfies(A,S):- depends(A,B,S), satisfies(B,S).

has_per(A,S1,N):- has_per(A,S,N), subservice(S1,S).
fulfills(A,S):- AND_decomp2(S,S1,S2), fulfills(A,S1),
                fulfills(A,S2).
fulfills(A,S):- OR_subservice(S1,S), fulfills(A,S1).
satisfies(A,S):- AND_decomp2(S,S1,S2), satisfies(A,S1),
                 satisfies(A,S2).
satisfies(A,S):- OR_subservice(S1,S), satisfies(A,S1).
```

Fig. 19 Axioms in the DLV system

```
:- requests(A,S), not satisfies(A,S).

:- delegate(perm,A,B,S,N), not has_per(A,S,N).
:- delegate(perm,A,B,S,N), not trustChain(perm,A,B,S,N).
:- has_per(B,S,N), owns(A,S), not trustChain(perm,A,B,S,N), A<>B.
:- fulfill(B,S), owns(A,S), not trustChain(perm,A,B,S,N), A<>B.
:- fulfills(A,S), not has_per(A,S,N).
```

Fig. 20 Properties in the DLV system

```
has_per(B,S,P):- #succ(P,N),delegate(perm,A,B,S,N),has_per(A,S,M),
                 not prohibitionChain(Pat,Cli,Rec), M>=N, N>1.
has_per(B,S,M):- delegate(perm,A,B,S,N), has_per(A,S,M),
                 not prohibitionChain(Pat,Cli,Rec), N>M, N>1.
delDChain(A,B,S,N):- delDenial(A,B,S,N).
delDChain(A,C,S,P):- #succ(P,N), delDChain(A,B,S,N),
                     delDChain(B,C,S,M), M>=N, N>1.
delDChain(A,C,S,M):- delDChain(A,B,S,N), delDChain(B,C,S,M),
                     N>M, N>1.
prohibitionChain(A,B,S):- prohibition(A,B,S).
prohibitionChain(A,C,S):- delDChain(A,B,S,N), prohibition(B,C,S),
                          N>0.
```

Fig. 21 Axioms with negative authorizations in the DLV system

Notice that axioms Ax13 and Ax15 in Fig. 16 cannot be directly implemented in DLV system since this requires a priori knowledge of the number of AND-subservices in which a service is decomposed. Thus, we introduce the predicate $\mathsf{AND_decomp}_n$ where n is the number of subservices in which a service is AND-decomposed. For instance, if service s is AND-decomposed into subservices s_1 and s_2, we use $\mathsf{AND_decomp}_2$ (s, s_1, s_2). Then, when we map the formal framework in DLV system, we need a "copy" of axioms Ax13 and Ax15 for each predicate $\mathsf{AND_decomp}_n$. For lack of space, in Fig. 19 we show only the case where n is equal to 2.

Figure 21 shows the implementation of axioms into DLV system when negative authorization are considered in the framework.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of databases. Addison-Wesley, Reading (1995)
2. Agrawal, R., Kiernan, J., Srikant, R., Xu, Y.: Hippocratic databases. In: Proceedings of VLDB'02, pp. 143–154. Morgan Kaufmann (2002)
3. Anderson, R.: A security policy model for clinical information systems. In: Proceedings of Symposium on Security and Privacy, pp. 30–43. IEEE Press (1996)
4. Antón, AI., Earp, JB.: A requirements taxonomy for reducing Web site privacy vulnerabilities. Requirements Eng. J. **9**(3), 169–185 (2004)
5. Axelrod, R.: The evolution of cooperation. Basic Books, London (1984)
6. Barnes, LB.: Managing the paradox of organizational trust. Harvard Bus. Rev. 59(2), 107–116 (1981)
7. Basin, D., Doser, J., Lodderstedt, T.: Model driven security: from UML models to access control infrastructures. TOSEM **15**(1), 39–91 (2006)
8. Blaze, M., Feigenbaum, J., Ioannidis, J., Keromytis, A.D.: The role of trust management in distributed systems security. secure internet programming **1603**, 185–210 (1999)
9. Blomqvist, K., Ståhle, P.: Building organizational trust. In: proceedings of 16th Annual IMP Conf. (2000)
10. Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., Perini, A.: TROPOS: An agent-oriented software development methodology. JAAMAS **8**(3), 203–236 (2004)
11. Castelfranchi, C., Falcone, R.: Principles of trust for MAS: Cognitive anatomy, social importance and quantification. In: proceedings of ICMAS'98, pp. 72–79. IEEE Press (1998)

12. Chu, Y.H., Feigenbaum, J., LaMacchia, B., Resnick, P., Strauss, M.: REFEREE: Trust management for web applications. computer networks and ISDN Systems **29**(8–13), 953–964 (1997)
13. Chung, L., Nixon, B.: Dealing with non-functional requirements: three experimental studies of a process-oriented approach. In: Proceedings of ICSE'95, pp. 25–37. ACM Press (1995)
14. Damianou, N.: A policy framework for management of distributed systems. Ph.D. Thesis, University of London (2002)
15. Devanbu, P.T., Stubblebine, S.G.: Software engineering for security: a roadmap. In: Proceedings. of ICSE'00 - Future of Software Engineerring Track, pp. 227–239 (2000)
16. Ebert, C.: Requirements BEFORE the requirements: understanding the upstream Impacts. In: Proceedings of RE'05, pp. 117–124. IEEE Press (2005)
17. Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B., Ylonen, T.: Simple public key certificates. Internet Draft (work in progress) (1999)
18. Giorgini, P., Massacci, F., Mylopoulos, J.: Requirement engineering meets security: a case study on modelling secure electronic transactions by VISA and Mastercard. In: Proceedings. of ER'03, *LNCS 2813*, pp. 263–276. Springer, Berlin Heidelberg Newyork (2003)
19. Giorgini, P., Massacci, F., Mylopoulos, J., Zannone, N.: Filling the gap between requirements engineering and public key/trust management infrastructures. In: Proceedings. of EuroPKI'04, *LNCS 3093*, pp. 98–111. Springer, Berlin Heidelberg Newyork (2004)
20. Giorgini, P., Massacci, F., Mylopoulos, J., Zannone, N.: Requirements engineering meets trust management: model, methodology, and reasoning. In: Proceedings. of iTrust'04, *LNCS 2995*, pp. 176–190. Springer, Berlin Heidelberg Newyork (2004)
21. Giorgini, P., Massacci, F., Mylopoulos, J., Zannone, N.: Modeling security requirements through ownership, permission and delegation. In: Proceedings of RE'05, pp. 167–176. IEEE Press (2005)
22. Giorgini, P., Massacci, F., Mylopoulos, J., Zannone, N.: Modelling social and individual trust in requirements engineering methodologies. In: Proceedings of iTrust'05, *LNCS 3477*, pp. 161–176. Springer, Berlin Heidelberg Newyork (2005)
23. Jim, T.: SD3: a trust management system with certified evaluation. In: Proceedings of Symposium on Security and Privacy, pp. 106–115. IEEE Press (2001)
24. Jürjens, J.: Secure Systems Development with UML. Springer, (2004)
25. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for knowledge representation and reasoning. TOCL (2005)
26. Li, N., Grosof, B.N., Feigenbaum, J.: Delegation logic: a logic-based approach to distributed authorization. TISSEC **6**(1), 128–171 (2003)
27. Li, N., Mitchell, J.C., Winsborough, W.H.: Design a role-based trust-management framework. In: Proceedings of Symposium on Security and Privacy, pp. 114–130. IEEE Press (2002)
28. Liu, L., Yu, E.S.K., Mylopoulos, J.: Security and Privacy Requirements Analysis within a Social Setting. In: Proceedings of RE'03, pp. 151–161. IEEE Press (2003)
29. Massacci, F., Mylopoulos, J., Zannone, N.: From hippocratic databases to secure tropos: a computer-aided re-engineering Approach. IJSEKE (2006). (in press).
30. Massacci, F., Penserini, L. (eds.): In: Proceedings of Symposium on Requirements Engineering for Information Security (2005)
31. Massacci, F., Prest, M., Zannone, N.: Using a security requirements engineering methodology in practice: the compliance with the italian data protection legislation. Comp. Stand. Inter. **27**(5), 445–455 (2005)
32. Massacci, F., Zannone, N.: Detecting conflicts between functional and security requirements with secure tropos: John Rusnak and the allied irish bank. Tech. Rep. DIT-06-002, University of Trento (2006)
33. McDermott, J., Fox, C.: Using abuse case models for security requirements Analysis. In: Proceedings of ACSAC'99, pp. 55–66. IEEE Press (1999)
34. McKnight, D.H., Chervany, N.L.: The meanings of trust. Tech. Rep. 96-04, MIS Research Center (1996)
35. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. IEEE Comp. **29**(2), 38–47 (1996)
36. Sindre, G., Opdahl, A.L.: Eliciting security requirements with misuse cases. Requirements Eng. J. **10**(1), 34–44 (2005)
37. Sommerville, I.: Software engineering. Addison-Wesley, Reading (2001)
38. Toval, A., Olmos, A., Piattini, M.: Legal requirements reuse: a critical success factor for requirements quality and personal data protection. In: Proceedings of RE'02, pp. 95 –103. IEEE Press (2002)
39. Tryfonas, T., Kiountouzis, E., Poulymenakou, A.: Embedding security practices in contemporary information systems development approaches. Inform. Manage. Comp. Sec. **9**, 183–197 (2001)
40. van Lamsweerde, A., Brohez, S., De Landtsheer, R., Janssens, D.: From system goals to intruder anti-goals: attack generation and resolution for security requirements engineering. In: Proceedings of RHAS'03, pp. 49–56 (2003)
41. Yu, E., Cysneiros, L.: designing for privacy and other competing requirements. In: Proceedings of SREIS'02 (2002)
42. Yu, E.S.K.: Modelling strategic relationships for process reengineering. Ph.D. thesis, University of Toronto (1996)