Security and Trust Requirements Engineering*

Paolo Giorgini, Fabio Massacci, and Nicola Zannone

Department of Information and Communication Technology
University of Trento - Italy
{giorgini,massacci,zannone}@dit.unitn.it

Abstract. Integrating security concerns throughout the whole software development process is one of today's challenges in software and requirements engineering research. A challenge that so far has proved difficult to meet.
The major difficulty is that providing security does not only require to solve technical problems but also to reason on the organization as a whole. This makes the usage of traditional software engineering methologies difficult or unsatisfactory: most proposals focus on protection aspects of security and explicitly deal with low level protection mechanisms and only an handful of them show the ability of capturing the high-level organizational security requirements, without getting suddenly bogged down into security protocols or cryptography algorithms.
In this paper we critically review the state of the art in security requirements engineering and discuss the motivations that led us to propose the Secure Tropos methodology, a formal framework for modelling and analyzing security, that enhances the agent-oriented software development methodology i*/Tropos. We illustrate the Secure Tropos approach, a comprehensive case study, and discuss some later refinements of the Secure Tropos methodology to address some of its shortcomings. Finally, we introduce the ST-Tool, a CASE tool that supports our methodology.

1 Introduction

The last decades have seen an increasing awareness that security plays a key role in system development. Unfortunately, security modelling and policy work has been largely independent of system requirements and system models. The usual approach towards the inclusion of security within a system is to identify security requirements after system design. This is a critical problem [4], mainly because security mechanisms have to be fitted into a pre-existing design which may not be able to accommodate them [53]. Moreover, the implementation of the software system may assume security mechanisms that are simply not necessary. 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. Late analysis of security requirements can also generate conflicts between security needs and functional requirements of the system. Even with the growing interest in secure engineering, current methodologies for software (notably, information system) development do not address security-related problems [56], and fail to integrate successfully security concerns throughout the whole

* This article provides a survey of the research material which is described in [25–28].

development process. There has also been lack of interaction between researchers working on requirements modelling and security policy. Security is compromised most often not by breaking mechanisms such as encryption or security protocols, but by exploiting weaknesses in the way they are being utilized. Security mechanisms cannot be blindly inserted into a security-critical system. Instead, the overall system development process must take security concerns into account.

One of the current research challenge is to integrate security requirements analysis with the standard requirements process. A security requirements is a manifestation of a high-level organizational policy into the detailed requirements of a specific system. The integration of security engineering into a model-driven software development approach has advantages. Security requirements can be formulated and integrated into system designs at a high level of abstraction. In this way, it becomes possible to develop security aware applications that are designed with the goal of preventing violations of a security policy. At one side of the spectrum, the call for SE profession has been on good coding practices to avoid errors that could compromise the software's security (e.g., [60]). At the other extreme, the emphasis has been on securing the organization and its procedures (e.g., [4]).

Across the whole spectrum among these two extremes, modelling and analysis of security requirements has become a key challenge for Software Engineering [14, 16], and it is the subject of this paper. In the next section (§2) we start a critical review of the existing proposals for security requirements engineering. Then, we discussed the key intuitions that lead us to propose an enhanced methodology and provide a description of Tropos concepts and describe the basic ones that we use for modelling security (§3). We introduce a scenario used as running example throughout the paper (§4). Then, we present a conceptual refinement of the framework (§5). Next, we formalize the security notions introduced in previous sections and define axioms and properties (§6). Finally, we introduce a CASE tool supporting our methodology (§7).

2 Security Requirements Engineering: A Survey

Strictly speaking of Software Engineering, modelling requirements is one of the key challenges that secure systems must meet (See Devanbu and Stubblebine's paper at ICSE [16]) and a number of researchers have been heeding the call. Proposals for Security Requirements Engineering can be classified under one of two classes: object-level and meta-level modelling.

The **object-level modelling** uses an off-the-shelves requirement framework, such as UML, KAOS, i*/Tropos, etc. and model in that framework a number of security requirements. The analysis features of the framework are then used to draw conclusions about the security modelling or to derive some guidance for the implementation.

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 modelling framework are immediately usable. If the framework is equipped with a formal semantics and formal reasoning procedures they are also inherited. In the formal framework the "security-notions" are indistinguishable from other objects, i.e., 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 modelling of general relationships or rules (such as all processing of personal data should be authorized by the person whose data is being processed).

In the early requirement arenas we can list a number of works in the object-level field. For instance, in [62] security is frequently considered as a vague goal to be satisfied, while a precise description and enumeration of specific security properties and behavior is still missing. The work by Liu et al. [41] uses i*/Tropos for dealing with security and privacy requirements by introducing softgoal[1], as "Security" or "Privacy", to model these notions, and use dependencies analysis to check if the system is secure. In [5, 6], general taxonomies for security and privacy are established. These can serve as a general knowledge repository for a knowledge-based goal refinement process. Another early RE example is [55], which presents a requirements process model, based upon reuse, together with a reusable template to organize security policies in a organization and a catalog filled with reusable personal data security requirements. Finally, He et al. [32] present a goal-driven framework for modelling privacy requirements in the role engineering process. The goal of this framework is to bridge the gap between competing stakeholders' security and privacy requirements, i.e., companies' privacy practices may be in conflict with user preferences. Privacy requirements are modelled as contexts and constraints of permissions and roles.

The **meta-level modelling** takes a off-the-shelves requirement framework as well as object-level modelling approach, but enhance it with linguistic constructs that capture security requirements. The analysis feature or implementation guidance of the framework must then be revised to allow for the new features.

The meta-level models trade off readiness for expressivity and compactness. The addition of suitable constructs makes usually 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 doesn't use the new features then one can still inherit all the old framework capabilities.

The need for conceptual models of security features have brought up a number of proposals especially in UML community. In approaches explicitly intended for security, we find the CORAS methodology for modelling risk and vulnerability [19]. Jürjens proposes UMLsec [35], an extension of the Unified Modelling Language (UML), for modelling security related features, such as confidentiality and access control. He proposes a concept for specifying requirements on confidentiality and integrity in analysis models based on UML. Lodderstedt et al. [42] present a UML-based modelling language (SecureUML). Their approach is focused on modelling access control policies and integrating them into a model-driven software development process. SecureUML is a modelling language designed to integrate information relevant to access control into application models defined with UML. The language builds on the access control

[1] Mostly non-functional requirements was satisfaction is fuzzy.

model of RBAC [18, 33, 47, 51] with additional support for specifying authorization constraints.

To address security concerns during software design, Doan et al. [17] incorporate Mandatory Access Control (MAC) into UML. Ray et al. [49] propose to model RBAC as a pattern by using UML diagram template. Further, they represent constraints on RBAC model through the Object Constraint Language. One of the major limitations of all these proposals is that they treat security in system-oriented terms, and do not support the modelling and analysis of security requirements 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 security engineering we need to model the organization and social relationships between all actors involved in the system.

For early requirements, a preliminary modification of Tropos methodology has been proposed in [45]. In particular, this extension use security constraints and secure capabilities as basic concepts. However, [23] shows that the key missing concept is the separation of the notions of offering a service and ownership of the very same service. Further, it does not allow for the modelling of trust relationships.

Other approaches propose to model the behavior of attackers. Crook et al. [14] introduce the notion of anti-requirements to represent the requirements of malicious attackers. Anti-requirements are expressed in terms of the problem domain phenomena and are quantified existentially: an anti-requirement is satisfied when the security threats imposed by the attacker are realized in any one instance of the problem. Lin et al. [40] incorporate anti-requirements into abuse frames. The purpose of abuse frames is to represents security threats and to facilitate the analysis of the conditions in the system in which a security violation occurs. They allow the examination of a system's vulnerabilities to different kinds of security threats in a bounded context. Abuse frames share the same notation as the normal problem frames, but each domain is now associated with a different meaning. McDermott and Fox adapt use cases [44] to capture and analyze security requirements, and they call the adaption an abuse case model. An abuse case is an interaction between a system and one or more actors, where the results of the interaction are harmful to the system, or one of the stakeholders of the system. Guttorm and Opdahl [52] propose to model security by defining misuse cases, the inverse of UML use cases, which describe functions that the system should not allow. This new construct makes it possible to represent actions that the system should prevent together with those actions which it should support. Moving towards early requirements, an extension of the KAOS framework is presented in [59] where the notion of obstacle is introduced. KAOS uses the notion of goal as a set of desired behaviors. Likewise, an obstacle defines a set of undesirable behaviors. Therefore, the negation of such obstacles is used to determine preconditions for the goal to be achieved. Although obstacle are sufficient for modelling accidental, non-intentional obstacles to security goals, they appear too limited for modelling and resolving malicious, intentional obstacles. To this end, van Lamsweerde et al. [58] introduce the notion of anti-requirements and anti-goals that are, respectively, the requirements of malicious attackers and the intentional obstacles to security goals.

2.1 Towards a "Terra Incognita": Why a New Methodology is Needed

Most proposals in the literature focus on protection aspects of security and explicitly deal with a series of security services (integrity, availability etc.) and related protection mechanisms (such as passwords, or cryptographic mechanisms). If we look at the requirement refinement process of many proposals, we find out that at 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 modelling process should instead makes it clear why encryption, access control and authentication are necessary. What is missing is capturing the high-level security requirements, without getting suddenly bogged down into security solutions or cryptographic algorithms.

Early requirements requires to reason about trust relationships, ownership and delegation of authority besides the traditional notion of functional dependencies. The first step in this direction is described the papers [25, 26] which extended the i*/Tropos modelling framework [10] to introduces concepts such as ownership, trust, and delegation within a requirements modelling framework and shows how security and trust requirements can be derived and analyzed.

After a large case study on the compliance of an ISO-17799-like security policy [43] with Italian privacy legislation, it was concluded that the concepts offered by Secure Tropos are the right ones but are too coarse-grained to capture important security facets.

The first observation is that for pragmatic reasons, it is often the case that services and permissions are delegated to actors who are not trusted. Nevertheless, the overall system is still considered secure if there is a way to hold such delegations accountable by monitoring their (wrong) doings.

The second observation is that trust in actors (or lack thereof) comes in different flavors: we may trust an actor to actually deliver the services we require (taking into account skills and/or commitment), or to honor granted permissions. In trust management and authorization settings (e.g. [7, 15, 38]) one only finds delegations of permission (through authorization). Requirements of availability are equally important, however, and can only be captured by modelling delegation of execution (where one actor delegates to another the responsibility to execute a service).

Finally, in a recent study, the majority of Information Security Administrators said that their biggest worry is employee negligence and abuse [48]. Internal attacks can be more harmful than external attacks since they are being performed by trusted users that can bypass access control mechanisms. So, we need models that compare the structure of the organization (roles and relations among them) with the concrete instance of the organization (agents playing some roles in the organization and relations among them). The original Tropos proposal involves two different levels of analysis: social and individual. In the organization level we analyze roles and positions of the organization, whereas in individual level the focus is on single agents. Of course there is no explicit separation between the two levels, and so Tropos is not able to maintain the consistency between the social level (roles and positions) and the individual level (agent).

3 Secure Tropos: a Goal Oriented SRE Methodology

Secure Tropos [25, 26] enhances the agent-oriented software development methodology i*/Tropos [10]. The Tropos methodology is intended to support all analysis and design activities in the software development process, from the application domain analysis down to the system implementation. In particular, Tropos rests on the idea of building a model of the system-to-be and its environment, that is incrementally refined and extended, providing a common interface to the various software development activities, as well as a basis for documentation and evolution of the software.

Tropos uses the concepts of actor, goal, plan, resource and social dependency for defining the obligations of actors (dependees) to other actors (dependers). A goal represents the strategic interests of an actor. A plan specifies a particular course of action that produces a desired effect, and can be executed in order to satisfy a goal. A resource represents a physical or an informational entity. Finally, a dependency between two actors indicates that one actor depends on another to accomplish a goal, execute a plan, or deliver a resource. Tropos is well suited to to describe both an organization and an IT system. As we already discussed, in [23] we have argued that it lacks the ability to capture at the same time the functional and security features of the organization, and hence the new proposal.

In the following, we introduce Secure Tropos as an extension of the requirements analysis phase of the Tropos Methodology. Basic concepts, relationships, and models will be presented along the methodological approach and the modelling activities.

3.1 Requirement analysis phase

Requirement analysis represents the initial phase in many software engineering methodologies. Similarly to other software engineering approaches, in Tropos the final goal of requirement analysis is to provide a set of functional and non-functional requirements for the system-to-be. The requirements analysis in Tropos is split in two main phases: *Early Requirements* and *Late Requirements* analysis. Both share the same conceptual and methodological approach. Thus most of the ideas introduced for early requirements analysis are used for late requirements as well.

More precisely, during the first phase, the requirements engineer identifies the domain stakeholders and models them as social actors, who depend on one another for goals to be achieved, plans to be performed, and resources to be furnished. By clearly defining these dependencies, it is then possible to state the *why*, beside the *what* and *how*, of the system functionalities and, as a last result, to verify how the final implementation matches the real needs.

In the *Late Requirements* analysis, the conceptual model is extended including a new actor, which represents the system, and a number of dependencies with other actors part of the environment. These dependencies define all the functional and non-functional requirements of the system-to-be.

3.2 The key concepts

Models in Tropos are acquired as instances of a *conceptual metamodel* resting on the following concepts/relationships:

– **Actor**, which models an entity that has strategic goals and intentionality within the system or the organizational setting. An actor represents a physical or a software *agent* as well as a *role* or *position*. While we assume the classical AI definition of software agent, that is, a software having properties such as autonomy, social ability, reactivity, proactivity, as given, for instance in [46], in Tropos we define a *role* as an abstract characterization of the behavior of a social actor within some specialized context or domain of endeavor, and a *position* represents a set of roles, typically played by one agent. An agent can occupy a position, while a position is said to cover a role. A discussion on this issue can be found in [61].
– **Goal**, which represents actors' strategic interests. We distinguish hard goals from softgoals, the second having no clear-cut definition and/or criteria for deciding whether they are satisfied or not. According to [12], this different nature of achievement is underlined by saying that goals are *satisfied* while softgoals are *satisficed*. Softgoals are typically used to model non-functional requirements.
– **Plan**, which represents, at an abstract level, a way of doing something. The execution of plan can be a means for satisfying a goal or for satisficing a softgoal.
– **Resource**, which represents a physical or an informational entity. The main difference with an agent is that a resource has not intentionality.
– **Dependency** between two actors, which indicates that one actor depends, for some reason, on the other in order to attain some goal, execute some plan, or deliver a resource. The former actor is called the *depender*, while the latter is called the *dependee*. The object around which the dependency centers is called *dependum*. In general, by depending on another actor for a dependum, an actor is able to achieve goals that it would otherwise be unable to achieve on its own, or not as easily, or not as well. At the same time, the depender becomes vulnerable. If the dependee fails to deliver the dependum, the depender would be adversely affected in its ability to achieve its goals.

Four new relationships have been introduced in Secure Tropos:

– **Ownership**, which indicates that the actor is the legitimate owner of some goal, some plan, or some resource. The owner has full authority concerning to achieve his goal, execute his plan, or use his resource, and he can also delegate this authority to other actors.
– **Provisioning**, which indicates that the actor has the capability to achieve some goal, execute some plan, or deliver a resource.
– **Trust**, between two actors, which indicates the believe of one actor that the other does not misuse some goal, some plan, or some resource. The former actor is called the *truster*, while the latter is called the *trustee*. The object around which the dependency centers is called *trustum*. In general, by trusting another actor for a trustum, an actor is sure that the trustum is properly used. At the same time, the truster becomes vulnerable. If the trustee misuses the trustum, the truster cannot guarantee to achieve some goal, execute some plan, or deliver a resource securely.
– **Delegation**, between two actors, which indicates that one actor delegates to the other the permission to achieve some goal, execute some plan, or use a resource. The former actor is called the *delegater*, while the latter is called the *delegatee*. The object around which the dependency centers is called *delegatum*. In general,

delegation marks a formal passage in the domain that is currently modelled by the requirements engineers. This would be matched by the issuance of a delegation certificate such as digital credential or a letter if we are delegating permission or by a call to an external procedure if we are delegating execution.

3.3 Modelling activities

Various activities contribute to the acquisition of a first early requirement model, to its refinement and to its evolution into subsequent models. They are:

- **Actor modelling**, which consists of identifying and analyzing both the actors of the environment and the system's actors and agents. In particular, in the early requirement phase actor modelling focuses on modelling the application domain stakeholders and their intentions as social actors which want to achieve goals. During late requirement, actor modelling focuses on the definition of the system-to-be actor.
- **Dependency modelling**, which consists of identifying actors which depend on one another for goals to be achieved, plans to be performed, and resources to be furnished. In particular, in the early requirement phase, it focuses on modelling goal dependencies between social actors of the organizational setting. New dependencies are elicited and added to the model upon goal analysis performed during the goal modelling activity discussed below. During late requirements analysis, dependency modelling focuses on analyzing the dependencies of the system-to-be actor. In the architectural design phase, data and control flows between sub-actors of the system-to-be actors are modelled in terms of dependencies, providing the basis for the capability modelling that will start later in architectural design together with the mapping of system actors to agents.

A graphical representation of the model obtained following these modelling activities is given through *actor diagrams*, called *dependency model*, which describe the actors (depicted as circles), their goals (depicted as ovals and cloud shapes) and the network of dependency relationships among actors (two arrowed lines connected by a graphical symbol varying according to the dependum: a goal, a plan or a resource).

- **Goal and plan modelling** rests on the analysis of an actor goals, conducted from the point of view of the actor, by using three basic reasoning techniques: *means-end analysis*, *contribution analysis*, and *AND/OR decomposition*. In particular, means-end analysis aims at identifying plans, resources and softgoals that provide means for achieving a goal. Contribution analysis identifies goals that can contribute positively or negatively in the fulfillment of the goal to be analyzed. In a sense, it can be considered as an extension of means-end analysis, with goals as means. AND/OR decomposition combines AND and OR decompositions of a root goal into subgoals, modelling a finer goal structure. Goal modelling is applied to early and late requirement models in order to refine them and to elicit new dependencies. During architectural design, it contributes to motivate the first decomposition of the system-to-be actors into a set of sub-actors.

A graphical representation of goal and plan modelling is given through *goal diagrams*, which appears as a balloon within which goals of a specific actor are analyzed and dependencies with other actors are established. Goals are decomposed into subgoals and positive/negative contributions of subgoals to goals are specified. Goal decomposition can be closed through a means-end analysis aimed at identifying plans, resources and softgoals that provide means for achieving the goal.

The revised methodology introduces new steps that replaces the old ones:

- **Trust modelling** which consists of identifying actors which trust other actors for goal, plans, and resources, and actors which own goal, plans, and resources. In particular, in the early requirement phase, it focuses on modelling trust relations between social actors of the organizational setting. New trust relations are elicited and added to the model upon the refinement activities discussed above. During late requirements analysis, trust modelling focuses on analyzing the trust relations of the system-to-be actor.
- **Delegation modelling** which consists of identifying actors which delegate to other actors the permission and task of execution on goals, plans, and resources. In particular, in the early requirement phase, it focuses on modelling delegations between social actors of the organizational setting. New delegations are elicited and added to the model upon the refinement activities discussed above. During late requirements analysis, delegation modelling focuses on analyzing the delegations involving the system-to-be actor.

A graphical representation of the models obtained following these last two modelling activities is given through two different kinds of actor diagrams: *trust model*, and *trust management implementation*. Essentially, the first represents the trust network among the actors involved in the system and the latter represents which permissions are effectively delegated by actors and which actors receive such permissions. These models use the same notation for actors, goals, plans and resource used during dependency modelling. The old dependency model is replaced by the delegation of execution model.

3.4 Process

The overall methodological process is an iterative process in which the above presented modelling activities are used to produce different kinds of actor and goal diagrams. Table 1 summarizes the process activities and the diagrams elaborated in each activity. The diagrams produced in one activity are used as input for the other activities.

The process starts with the actor modelling activity (1) in which the relevant actors (stackholders and existing software (sub)systems) are elicited and modelled with their goals. The actor diagram produced after this activity is used as input for the dependency modelling activity (2old), where the dependencies between the actors are discovered and established. The resulting actor diagram can be either used to further revise the initial actor diagram or as input for the next activity (3). Goal modelling focus on the goals associated to each actor of the actor diagram and it analyzes them using various forms of analysis as described earlier. During the analysis new dependencies can be discovered

Activity	Diagrams produced
1. Actor modelling	Actor diagram: actors and their goals are elicited
2old. Dependency modelling	Actor diagram: dependencies between actors are discovered
2a. Trust modelling	Trust diagram: trust relationships between actors are discovered
2b. Delegation modelling	Trust management implementation diagram: delegations between actors are modelled
3. Goal modelling	Goal diagram: actor goals are analyzed
4. Plan modelling	Goal diagram: plans associated to goals are analyzed

Table 1. Activities and diagrams produced during the analysis process

so to revise and enrich the model produced in (2). Goal diagram is also used as input for plan modelling activity (4), where each single goal is analyzed in terms of plans that can be used for its fulfillment. Plans are analyzed in details and new dependencies between actor can emerge so to require a new dependency analysis (2old). In the new model we start with the trust model (2a), which in turn can require a further goal analysis (3). Dependency can then be devised by as in step (2old) by modelling delegation and trust (if any). This may require further goal analysis (3) as in the standard Tropos project. The final trust model is used to develop the trust management implementation for permission (2b), that finally can be used to revise the delegation of execution model (2b) and the trust model (2a). The process ends when no further analysis are needed.

4 Using SRE for Compliance with Data Privacy Legislation

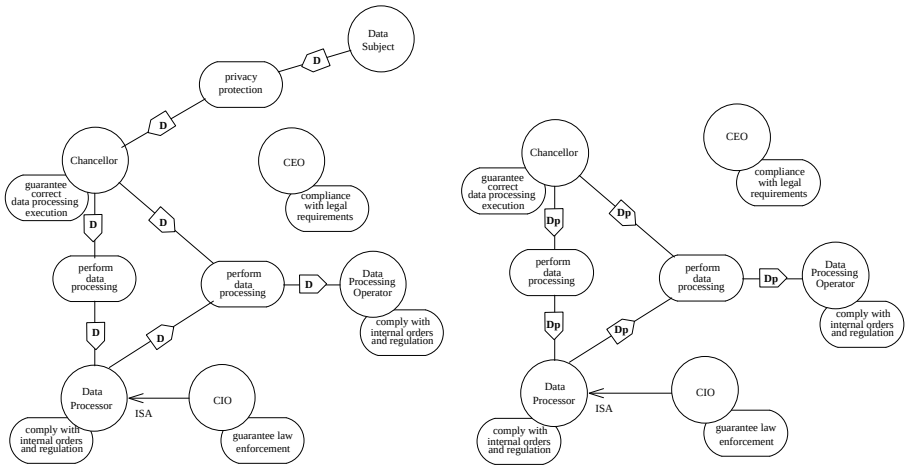
To instantiate some of the above mentioned concepts we show some fragments of a complex case study: the compliance to the Italian legislation on Privacy and Data Protection by the University of Trento, leading to the definition and analysis of an ISO-17799-like security management scheme (we refer to [43] for more detials). The final EU and Italian legislation systematized the norms on privacy and data protection by specifying

- the definitions of personal data, sensitive data, and data processing,
- the definitions of all entities involved in data processing, their roles and responsibilities (controller, processor, operator, subject),
- the obligations relating to public and private data controllers with specific reference to the legitimate purpose of data processing and the adoption of minimal precautionary security measures to minimize the risks on data.

4.1 Modelling Actors

The first activity in the early requirements phase is actors' modelling. In our example we can list some of them:

Data Controller determines the purposes and means of the processing of personal data. In the University, the data controller is identified with Chancellor (as the post-holder is also the legal representative of the University).



(a) Actor Diagram

(b) Trust Management Implementation diagram

Fig. 1. Actor Diagrams

Data Processor monitors personal data processing on behalf of the controller. In the University, these are:
- Faculty Deans;
- Head of Department;
- Central Directorate Managers, and in particular with:
  - Chief Executive Officer (CEO);
  - Chief Information Officer (CIO).

Data Processing Operator is appointed by the data controller or processor to perform the operations related to the data processing or to manage and maintain the information systems and services. At University of Trento, these are:
- Personal Data Processing Operator;
- Database Security Operator;
- Network Security Operator.

Data Subject is the natural or legal person to whom the personal data are related. In the Secure Tropos terminology, this is the legitimate owner of the data.

4.2 Modelling Dependencies and Delegation

The analysis proceeds introducing the functional dependencies and the delegation of permission between actors and the consequent integrated security and functional requirements. Figure 1(a) and Figure 1(b) show the functional dependency model and the trust management implementation. We use delegation of permission (**Dp**) to model the actual transfer of rights in some form (e.g. a digital certificate, a signed paper, etc.), and **D** for functional dependency.

In the functional dependency model, *Chancellor* is associated with a single relevant goal: *guarantee correct data processing execution*, while *CEO* has an associated goal *compliance with legal requirements*. Along similar lines, *Data Processor* and

Data Processing Operator want to *comply with internal orders and regulation*, while *CIO*, wants to *guarantee law enforcement*. Finally, the diagram includes some functional dependencies: *Data Subject* depends on *Chancellor* for *privacy protection* goal; *Chancellor* depends on *Data Processor* and *Data Processing Operator* to *perform data processing*; and, in turn, *Data Processor* depends on *Data Processing Operator* for it.

In the actor diagram, *Chancellor* is associated with a single relevant goal: *guarantee correct data processing execution*, while *CEO* has an associated goal *compliance with legal requirements*. Along similar lines, *Data Processor* and *Data Processing Operator* want to *comply with internal orders and regulation*, while *CIO*, wants to *guarantee law enforcement*. Finally, the diagram includes some delegations of execution: *Data Subject* delegates to *Chancellor* the goal *privacy protection*; *Chancellor* delegates to *Data Processor* and *Data Processing Operator* the goal *perform data processing*; and, in turn, *Data Processor* delegates it to *Data Processing Operator*.

In the trust management implementation diagram, *Chancellor* delegates permissions to *perform data processing* to *Data Processor* and *Data Processing Operator*. In turn, *Data Processor* delegates permissions to *perform data processing* to *Data Processing Operator*.

At this stage, the analysis already reveals a number of pitfalls in the actual document template provided by the ministry's agency. The most notable one is the absolute absence of functional dependencies between the Chancellor and the CEO, who is actually the one who runs the administration. Such functional dependency is present in the Universities statutes, but not here (an apparently unrelated document).

Another missing part in the trust management implementation is the delegation of permission from the data subject. This can be also automatically spotted with the techniques developed in [26]. Somehow paradoxically (for a document template enacted in fulfillment of a Data Protection Act) the process of acquisition of data (and the relative authorization) is neither mentioned nor forseen. In practice this gap is solved by the University by a blanket authorization: in all the paper or electronic data collection steps a signature is required to authorize the processing of data in compliance with the privacy legislation.

4.3 Goal Refinement

In this paper, we present a goal analysis for *Data Processor* and refer to [43] for an accurate analysis of the other actors involved in the system.

Figure 2 shows the goal analysis for Data Processor, relative to the goal *comply with internal orders and regulation*. This goal is decomposed into *provide for appointing data processing operators*, *security control* and *adopt security measures* for which *Data Processor* depends on *CIO* and *CEO*. The goal *provide for appointing data processing operators* is decomposed into three goals: *identify data processing operators* for which *Data Processor* depends on *Data Processing Operator*, *give instructions to data processing operators* for which *Data Processing Operator* depends on *Data Processor*, and *enable access profile* for which *Data Processing Operator* depends on *Data Processor* and, in turn, *Data Processor* depends on *CIO*. The goal *enable access profile* is decomposed into *assign access profile*, *assign ID and password* which *Data Processor* depends on *CIO*, and *communicate name of security operators* for which

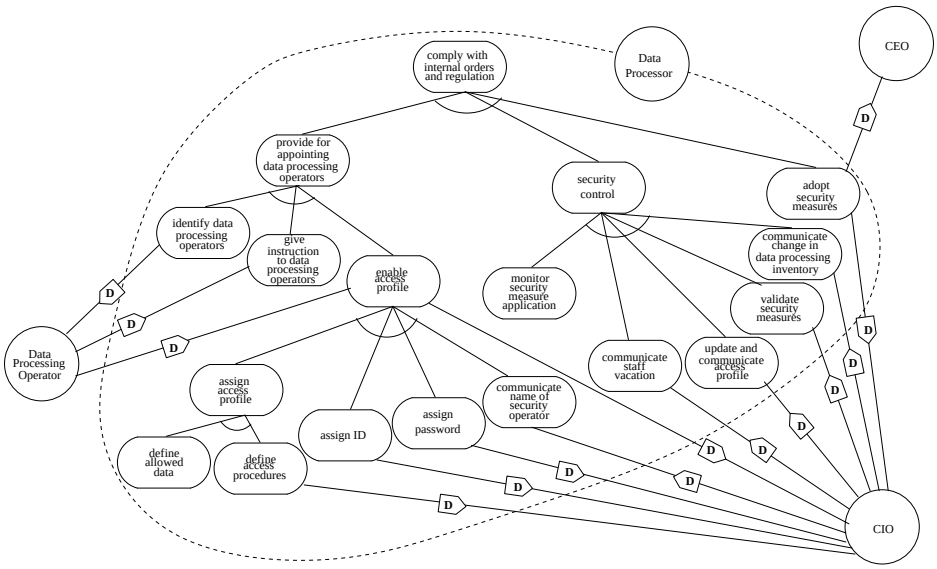


Fig. 2. Functional Dependency Model for Data Processor

CIO depends on *Data Processor*. The goal *security control* is decomposed into *monitor security measure application* and other goals, such as *communicate staff vacation* and *update and communicate access profile*, for which *CIO* depends on *Data Processor*, *update and communicate access profile*.

Figure 3 shows the trust management implementation for *Data Processor*. The diagram displays that *Data Processor* delegates *mail with instructions* to *Data Processing Operator*. Further, *Data Processor* delegates the *list of name of security operators*, *list of employees in vacation*, *access profile* and *data processing inventory* to *CIO*. Finally, *Data Processor*receives from *CEO* the *list of security measures*.

5 The Plot Thickens: Refining Delegation and Trust

In this section, we introduce a conceptual refinement of the delegation and trust relationships, that will allow us to capture and model important security facets [27, 28].

In order to explain the conceptual refinement we will use examples based on the case study presented in previous section. For the sake of readability we introduce here dramatis personae [2] together with the rules they play:

Alice is an administrative officer, for example of the teaching evaluation office;
Bob, Bert, and Bill are students;
Sam is (the manager of) the student IT system;
Paul and Peter are professors.

[2] This impersonation is actually closer to reality than one may think: the law requires the assignment of responsibility of each IT sub-system to a person.

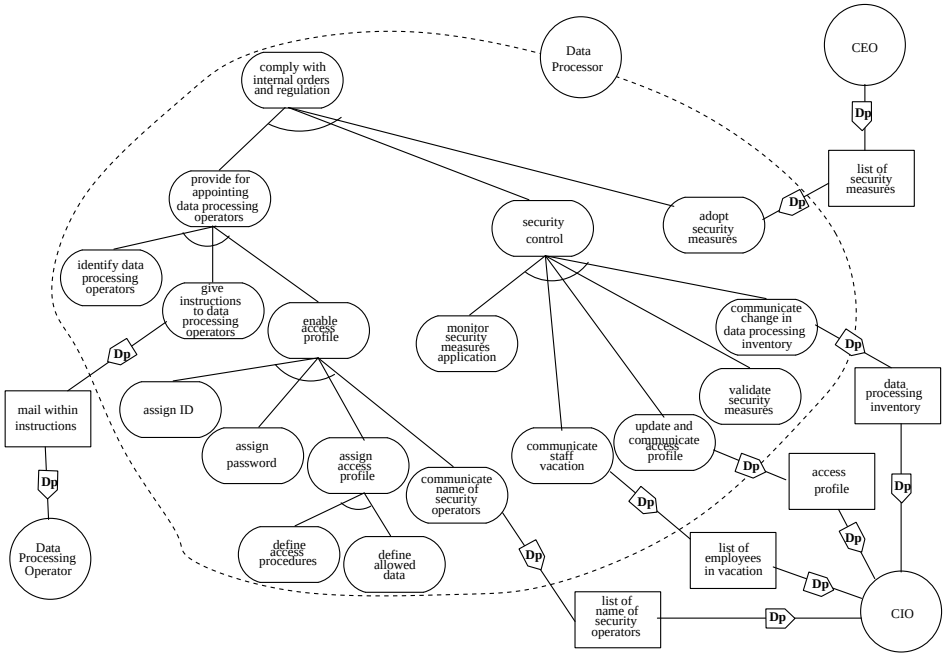


Fig. 3. Trust Management Implementation for Data Processor

5.1 Execution vs Permission

Example 1. Alice is interested in gathering data on students' performance, for which she depends on Sam. Bob owns his sensitive personal information, such as his student careers. Bob delegates permission to provide information about his career to Sam on condition that his privacy is protected (i.e., his identity is not revealed).

In this scenario, there is a difference of relationship between Alice–Sam and Bob–Sam. This difference is due to a difference in the type of delegation.

Example 2. Bob *delegates permission* to Sam to provide only the relevant information and nothing else. On the other hand, Alice, who wants student data, *delegates* the *execution* of her goal to Sam. According to Alice, Sam should at least fulfill the goal she requires. She is not interested in what Sam does with Bob's trust, apart from getting her information. The major worry of Alice is availability whereas Bob cares about authorization. In other words, Alice's major concerns would be that tasks are delegated to people that can actually do them, whereas Bob would be concerned that subtasks are given to trusted people who will not misuse the permissions they have acquired.

If we want to check functional and security requirements consistency, it is essential to distinguish between these two notions of delegation. We use **at-most delegation** when the delegater wants the delegatee at most achieves the goal, execute the plan, or furnishes the resource. This is *delegation of permission*, where the delegatee thinks

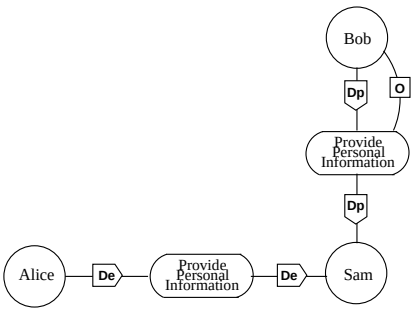
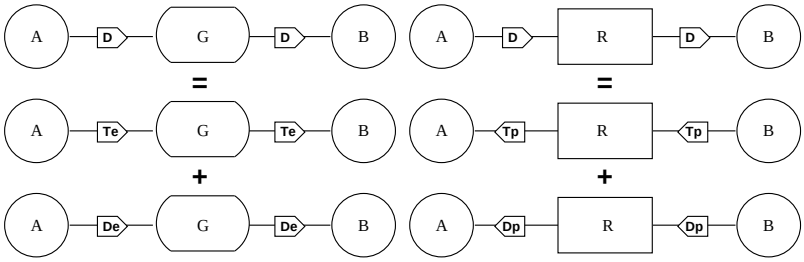


Fig. 4. At-least and At-most Delegation



(a) Goal Dependency (b) Resource Dependency

Fig. 5. Tropos dependency in terms of Secure Tropos

"I have the permission to achieve the goal (but I do not need to)", whereas **at-least delegation** means that the delegater wants the delegatee to achieve at least the goal. This is the *delegation of execution*. The delegatee thinks, "Now, I have to get the goal fulfilled (let's start working)". In the pictorial representation of Fig. 4 we represent these relationship as edges respectively labeled by **Dp** and **De**.

Further, we want to separate the concepts of trust and delegation, as we might need to model systems where some actors must delegate permission or execution to other actors they don't trust. Also in this case it is convenient to have a suitable distinction for trust in managing permission and trust in managing execution. The meaning of **at-most trust** is that an actor (truster) trusts that another actor (trustee) at most fulfills the goal but will not overstep it. The meaning of **at-least trust** is that an actor (truster) trusts that another actor (trustee) at least fulfills the goal.

Example 3. At-most trust is good for permissions: Bob trusts Sam to remain within certain bounds. He may delegate Sam more permissions than actually needed because Sam will not abuse them. At-least trust fits execution. Alice believe Sam can accomplish her plans and possibly more.

The new Secure Tropos concepts "explain" the classical Tropos dependency between two actors in terms of trust and delegation (Fig. 5).

Indeed, the semantics associated to the Tropos dependency states that there is an actor, the dependee, that wants to achieve a specific goal (perform a task or have a

resource) and there is another actor, the depender, that is able to satisfy the goal (perform the task or deliver the resource). The two actors get an agreement and a goal (task or resource) dependency is established between the two. The implicit assumption is that after the agreement the depender will be responsible for the goal and will do the best to achieve it.

The distinction between *execution* and *permission* allows us to define a dependency in terms of trust and delegation. In particular, when the dependum is a goal or a plan we have delegation and trust of execution, whereas when the dependum is a resource we have delegation and trust of permission. In symbols:

$$\mathsf{depends}(A, B, S) \Longleftrightarrow \mathsf{delegate}(exec, A, B, S) \wedge \mathsf{trust}(exec, A, B, S) \qquad (1)$$

where S is a goal or a plan, and

$$\mathsf{depends}(A, B, S) \Longleftrightarrow \mathsf{delegate}(perm, ID, B, A)\, S \wedge \mathsf{trust}(perm, B, A, S) \qquad (2)$$

where S is a resource. A graphical representation of these formulas is given, respectively, in Fig. 5(a) and in Fig. 5(b). These diagrams use the label **D** for Tropos dependency and labels **Te** and **Tp**, respectively for trust of execution and trust of permission. Notice also from Fig. 5 that the same dependency is mapped into differently oriented relations at the lower level.

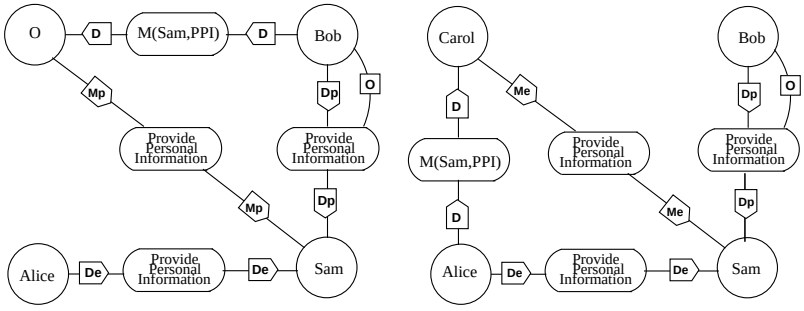
5.2 Introducing Distrust

Another refinement is the introduction of negative authorizations which are needed for some scenarios. Tropos already accommodates the notion of positive or negative contribution of goals to the fulfillment of other goals. We use negative authorizations to help the designer in shaping the perimeter of positive trust to avoid incautious delegation certificates that may give more powers than desired.

Suppose that an actor should not be entitled to achieve a goal, perform a plan, or delivery a resource. In situations where authorization administration is decentralized, an actor possessing the right to achieve a goal, execute a plan, or delivery a resource, can delegate the authorization to do that to the wrong actor.

We propose an explicit distrust relationship as an approach for handling this type of situations. This is also sound from a cognitive point of view if we follow the definition of trust given by [11]: trust is a mental state based on a set of beliefs. We can say that if, on your own knowledge, you feel to trust me, then you trust me. Similarly, if you feel like distrusting me, then you distrust me. Obviously, there are various reasons for distrusting agents such as unskillfulness, unreliability and abuse, but these situations are not treated here.

As we have done for trust, we also distinguish between distrust of execution and distrust of permission. The graphical diagrams presented in this paper use the labels **Se** and **Sp**, respectively, for distrust of execution and distrust of permission. In the case there is no explicit trust relationship between agents, the label "**?**" is used.



(a) At-most Monitor (b) At-least Monitor

Fig. 6. Monitoring

5.3 Monitoring

When work needs to be delegated even when there is no trust, then monitoring can offer a surrogate for trust. Accordingly to Gans's et al. [21], the existence of distrust can be tolerated with an additional overhead of monitoring the untrustworthy delegatee. Here we refine Gans's et al. intuition integrating it in our framework.

The goal of an actor playing the role of *monitor* is to check for the violation of trust[3]. The act of monitoring can be done by the delegater himself[4], or he can delegate it to some other actors to get it done. Depending on the type of delegation, we have two different kinds of monitors: **at-most monitor** and **at-least monitor**. Consider the situation presented in Fig. 4.

Example 4. Suppose that there is no trust between Bob and Sam for the goal "maintain privacy", but the student must delegate permission nonetheless. In this case, he depends (**D**) on the ombudsman (O) for monitoring if Sam transgresses her permissions. This is shown in Fig. 6(a)) with an at-most monitor (monitor for permission – **Mp**) relationship between the ombudsman and Sam.

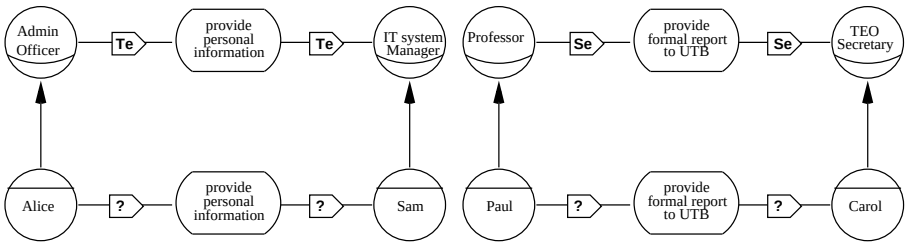
Example 5. If Alice is not confident that Sam will provide updated information, she may delegate to her secretary Carol the task of confirming with, or nagging Sam to insert new data as soon as it becomes available. This is shown in Fig. 6(b)) with an at-least monitor (monitor for execution – **Me**) relationship between Carol and Sam.

Another important distinction that emerges when we use a monitor is related to what we have to monitor. If we are monitoring a plan (i.e., a specific sequence of actions), the Monitor has to check if Sam executes the actions of the plan. What happens if Sam delegates the task or some of its subtasks to other actors?

Example 6. To achieve the goal delegated to him in Example 5, Sam will issue a letter to the head of each student secretariat office so that student marks are entered into the system within 30 days from the date that exams have taken place.

[3] Indeed, monitoring could also be used for the evaluation of the fulfillment of a goal assigned to a trusted actor.

[4] Intuitively, this is like saying that fellow is unreliable, I'll give him the job but keep an eye on him myself".



(a) Trust of execution (b) Distrust of execution

Fig. 7. Missing (dis)trust relations at individual level

A solution to this problem is to extend the monitoring to all sublevels of delegation until the level where the actual execution takes place. So, there will be a monitor relationship between the Monitor and all the actor involved in the execution of at least a part of the task.

Example 7. To reach the objective of 30 days requires that professors return to the office assigned marks. This is a further step of delegation of execution. Then, the actor responsible at the office, beside actually monitoring his employees, may also assign the task of reminding professors that they must return on time their mark sheets.

Notice that monitoring as such is not a primitive construct. It can be captured by other constructs within our modelling framework. Specifically, every goal, plan and resource will either be delegated during the design process to a trusted actor, or it will be delegated to an untrusted one, in which case the delegatee will be monitored by a trusted actor.

On the formal model this corresponds to a design pattern formalized in terms of additional axioms that allow us to conclude that an actor is confident that a goal will be executed, a plan will be performed or a resource will be furnished, or a permission will not be abused even if existing trust relations suggest otherwise.

Once we see monitoring as a simple design solution (essentially a security pattern) we can treat monitoring goals just as any other goal. So they can be further subject to refinement, delegation of execution and delegation of permission. Trust relationships linked to monitoring can then be captured with standard constructs. For example, monitoring often requires having permission to access monitored data or personnel. This itself may create problems of permission and authorization that can be model in the framework.

5.4 Social vs Individual Trust

When we model and analyze functional trust and security relationships, it is possible that such requirements are given only at individual level or at social level and that there is a mismatch between the levels. Let us see why this is needed with examples drawn from the same domain.

Example 8. According the University policy, administrative officers should trust managers of IT systems to get information they need to perform their duties (Fig. 7(a)). Sam is the new manager of the student IT system and Alice has never met him before. Still, Alice should trust Sam for getting student personal information in order to guarantee the availability of the goal.

Example 9. Professors should not rely on the teaching evaluation officer secretary for providing a formal report to the University Teaching Board (Fig. 7(b)). Here, Paul and Carol don't know each other. Then, Paul should distrust Carol for providing a formal report to University Teaching Board.

We don't consider the case in which the relations are missing at social level because this level represents the structure of the organization which should be described explicitly in the requirements. The presence of a large number of trust relations at individual level that is not matched by a social level may be an indicator of a missing link at social level (or of a problem in the organization for distrust relations). On the contrary, Hannoun et al. [31] propose to detect the inadequacy of an organization regarding the relations existing among the agents involved in the system.

In [26] we have only considered when trust is explicit, and we have not distinguished the case where there is explicit distrust and the case where no trust relation is given. Contrarily, in this paper we take in consideration all these three possibilities. The presence of positive and negative authorization at the same time could generate some conflicts on trust relationships. We define a *trust conflict* the situation where there are both a positive and a negative trust relation between two actors for the same trustum. Next, formal definitions are given.

Definition 1. *A conflict on trust of execution occurs when*

$$\exists x, y \in \texttt{Agent}\ \exists s \in \texttt{Goal} \cup \texttt{Task} \cup \texttt{Resource} \mid \mathsf{trust}(exec, x, y, s) \wedge \mathsf{distrust}(exec, x, y, s)$$

Definition 2. *A conflict on trust of permission occurs when*

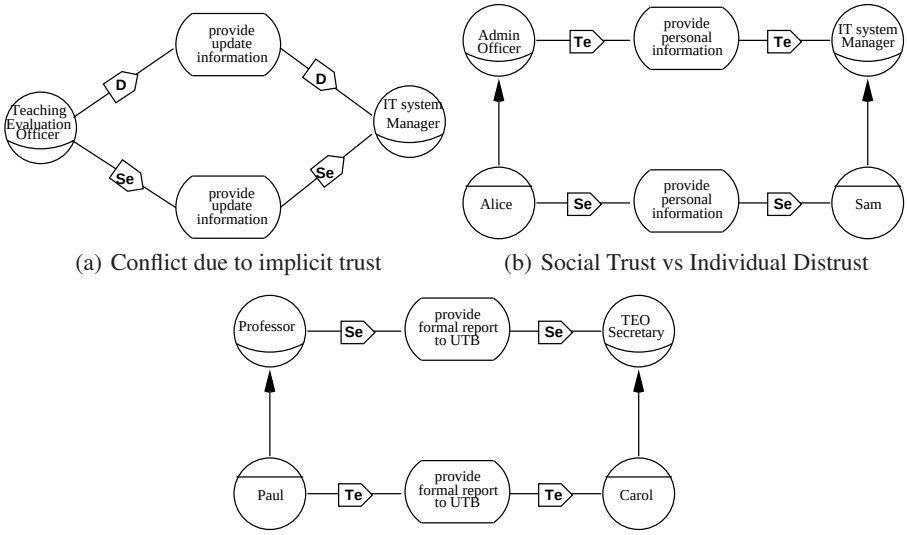
$$\exists x, y \in \texttt{Agent}\ \exists s \in \texttt{Goal} \cup \texttt{Task} \cup \texttt{Resource} \mid \mathsf{trust}(perm, x, y, s) \wedge \mathsf{distrust}(perm, x, y, s)$$

A trust conflict may exist, for example, since system designers wrongly put both a (implicit) trust relation and the corresponding distrust relation.

Example 10. The teaching evaluation officer depends on the manager of the student IT system for providing update information, but the latter is distrusted for such goal (Figure 8(a)).

When we model and analyze security requirements, it is also possible that such requirements are specified at both individual and social levels, they could be in contrast with each other.

Example 11. Consider again Example 8. What happens if Alice had some problems with Sam in the past and he doesn't trust her? This scenario is presented in Fig.8(b).



(a) Conflict due to implicit trust

(b) Social Trust vs Individual Distrust

(c) Social Distrust vs Individual Trust

Fig. 8. Conflicts on (dis)trust relations

Example 12. Consider again Example 9. What happen if Paul trusts Carol for providing a formal report to University Teaching Board? This scenario is presented in Fig. 8(c).

Monitoring, which we have introduced early in thic paper, is a good solution to this extent. So we don't need to add anything to the system just to cope with trust conflicts.

Example 13. Referring to Example 11, we believe that Alice should monitor (or delegate this task to another actor) whether Sam does what he has to do since the organization imposes her to trust him, but it is not her own choice.

6 Automated Reasoning in SRE

We use Datalog [1] as the underlying semantic framework, also to be close to the semantics of other frameworks for trust or security (e.g. [15, 39, 50]).

A Datalog 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. We use the notation $\{L\}$:-$L_1, \ldots, L_n$ to indicate that if $L_1, \ldots, L_n$ are true then L *may* be true. Essentially, L will be added to the model only if some constraints demand its inclusion. This construction can be captured with a simple encoding in logic programs. In Datalog, negation is treated as negation as failure: if there is no evidence that an atom is true, it is considered to be false. Hence if an atom is not true in some model, then its negation should be considered to be true in that model.

We start by presenting the predicates for our framework. We distinguish between two main types of predicates: extensional and intensional. Extensional predicates are

General predicates
goal(Goal : g)
plan(Plan : t)
resource(Resource : r)
agent(Agent : a)
position(Position : a)
role(Role : a)
play(Agent : a, Role : b)
is_a(Role : a, Role : b)
depends(Actor : a, Actor : b, Service : s)
delegate(Type : t, Actor : a, Actor : b, Service : s)
delegateChain(Type : t, Actor : a, Actor : b, Service : s)
trust(Type : t, Actor : a, Actor : b, Service : s)
trustChain(Type : t, Actor : a, Actor : b, Service : s)
distrust(Type : t, Actor : a, Actor : b, Service : s)
distrustChain(Type : t, Actor : a, Actor : b, Service : s)
monitoring(Type : t, Actor : a, Actor : b, Service : s)
confident(Type : t, Actor : a, Service : s)
Specific for execution
requests(Actor : a, Service : s)
provides(Actor : a, Service : s)
should_do(Actor : a, Service : s)
can_satisfy(Actor : a, Service : s)
Specific for Permission
owns(Actor : a, Service : s)
has_per(Actor : a, Service : s)
Goal refinement
subgoal(Service : s_1, Service : s_2)
OR_subgoal(Service : s_1, Service : s_2)
AND_subgoal(Service : s_1, Service : s_2)
AND_decomp(Service : s_1, Service : s_2, Service : s_3)

Table 2. Predicates

predicates set directly with the help of ground facts and are the ones corresponding the edge and circles drawn by the requirements engineer on the CASE tool. Intensional predicates are implicitly determined with the help of rules. Table 2 presents the predicates used to formalize the requirements. For compactness' sake we use the first argument of the predicates to indicate the type of actions. Thus, delegate, delegateChain, distrust, distrustChain, and monitoring have a type $t \in \{exec, perm\}$; trust, trustChain have a type $t \in \{exec, perm, mon\}$; and confident has a type $t \in \{satisfy, exec, owner, mon\}$. Once again, we specify predicates for generic "services" because differentiating them into goals, plans and resources is immediate[5].

The unary predicates goal, plan and resource are used respectively for identifying goals, tasks and resource. Note that type `Goal`, `Task` and `Resource` are sub-types of `Service`. We shall use letters S, G, T and R possibly with indices as metavariables

[5] For resources we must replace the subgoal relation with the part-of relation.

ranging over the terms, respectively, of type `Service`, `Goal`, `Task` and `Resource`. The intuition is that agent(a) holds if instance a is an agent, position(a) holds if instance a is a position, and role(a) holds if instance a is a role. Note that type `Agent`, `Position` and `Role` are sub-types of `Actor`. We shall use letters X, Y and Z as metavariables ranging over the terms of type `Actor`, A, B and C as metavariables ranging over the terms of type `Agent`, and T, Q and V as metavariables ranging over the terms of type `Role`. Metalevel variables are used as a syntactic sugar to avoid to write the predicates that type variables. For example, when the metavariable G occurs in a rule, the predicate goal(G) should be put in the body of the rule. The predicate play(a, b) holds if agent a is an instance of role b. The intuition is that is_a(a, b) holds if role a is a specialization of role b. The predicate depends(a, b, s) holds if actor a depends on actor b for service s. Notice also that when a relation uses variables of type `Actor` the relation can apply to both social and individual levels, but separately.

6.1 Formal Model for Execution

The predicates that we introduced correspond to the relations that the requirements engineer can actually draw during his analysis. The predicate requests(a, s) holds if actor a wants service s fulfilled, while provides(a, s) holds if actor a has the capability to fulfill service s. The predicate delegate($exec, a, b, s$) holds if actor a delegates[6] the execution of service s to actor b. Actor a is called *delegater*; actor b is called *delegatee*. The predicate trust($exec, a, b, s$) holds is actor a trusts that actor b at least fulfills service s. Actor a is called *truster*; actor b is called *trustee*. The predicate trust(mon, a, b, s) holds if actor a trusts that actor b monitors whether service s will be satisfied. The predicate monitoring($exec, a, b, s$) holds if actor a monitors if actor b at least can satisfy service s.

Other predicates are used to define properties that will be used during formal analysis. The predicates delegateChain($exec, a, b, s$) and trustChain($exec, a, b, s$) hold if there is a delegation and a trust chain respectively, between actor a and actor b. The predicate should_do(a, s) identifies actors who should directly fulfill the service. The basic idea of the predicate can_satisfy is that "for every goal I have assigned responsibilities so that it can be fulfilled". In other words, if an actor has the objective of fulfilling a service, he can satisfy it. Thus it locates the common leaves of the delegation trees of execution and permission. Thus, the predicate can_satisfy(a, s) holds if actor a can satisfy service s. The predicate confident($satisfy, a, s$) holds if actor a is confident that service s can be satisfied. Finally, we have the predicates for goal refinement. Their semantics and axiomatization are straight-forward.

The axiomatization is more complex for modelling execution as shown in Table 3. E1 and E2 build a delegation chain of execution. E3-8 define the intensional versions, trustChain and distrustChain of the extensional predicates trust and distrust that are used to build (dis)trust chains by propagating (dis)trust of execution (permission) relations. E5 and E6 (M1 and M2) build a trust chain for execution (monitoring); E5 builds

[6] For the sake of simplicity we do not deal with the question of depth here. See Li et al. [38] for an account of delegation with depth. What has emerged from several case studies is that depth is less important than qualifications such as "only to members of the same office".

Delegation
E1 $\mathsf{delegateChain}(exec, X, Y, S) \leftarrow \mathsf{delegate}(exec, X, Y, S)$
E2 $\mathsf{delegateChain}(exec, X, Z, S) \leftarrow \mathsf{delegate}(exec, X, Y, S) \wedge \mathsf{delegateChain}(exec, Y, Z, S)$
Trust
E3 $\mathsf{distrustChain}(exec, X, Y, S) \leftarrow \mathsf{distrust}(exec, X, Y, S)$
E4 $\mathsf{distrustChain}(exec, X, Z, S) \leftarrow \begin{cases} \mathsf{trustChain}(exec, X, Y, S) \wedge \mathsf{distrust}(exec, Y, Z, S) \wedge \\ \quad \mathsf{not\ distrustChain}(exec, X, Y, S) \end{cases}$
E5 $\mathsf{trustChain}(exec, X, Y, S) \leftarrow \mathsf{trust}(exec, X, Y, S) \wedge \mathsf{not\ distrustChain}(exec, X, Y, S)$
E6 $\mathsf{trustChain}(exec, X, Z, S) \leftarrow \begin{cases} \mathsf{trustChain}(exec, X, Y, S) \wedge \mathsf{trustChain}(exec, Y, Z, S) \wedge \\ \quad \mathsf{not\ distrustChain}(exec, X, Z, S) \end{cases}$
E7 $\mathsf{trustChain}(exec, X, Z, S) \leftarrow \mathsf{trustChain}(mon, X, Y, S) \wedge \mathsf{monitoring}(exec, Y, Z, S)$
E8 $\mathsf{trustChain}(exec, X, Y, S_1) \leftarrow \mathsf{subgoal}(S, S_1) \wedge \mathsf{trustChain}(exec, X, Y, S)$
M1 $\mathsf{trustChain}(mon, X, Y, S) \leftarrow \mathsf{trust}(mon, X, Y, S)$
M2 $\mathsf{trustChain}(mon, X, Z, S) \leftarrow \mathsf{trust}(mon, X, Y, S) \wedge \mathsf{trustChain}(mon, Y, Z, S)$
M3 $\mathsf{trustChain}(mon, X, Z, S) \leftarrow \mathsf{trustChain}(exec, X, Y, S) \wedge \mathsf{trustChain}(mon, Y, Z, S)$
M4 $\mathsf{trustChain}(mon, X, Y, S_1) \leftarrow \mathsf{subgoal}(S, S_1) \wedge \mathsf{trustChain}(mon, X, Z, S)$
Monitoring
M5 $\mathsf{monitoring}(exec, Y, Z, S_1) \leftarrow \begin{cases} \mathsf{delegateChain}(exec, X, Y, S_1) \wedge \\ \mathsf{monitoring}(exec, Z, X, S) \wedge \mathsf{subgoal}(S_1, S) \end{cases}$
M6 $\mathsf{confident}(mon, X, Y, S) \leftarrow \mathsf{trust}(mon, X, Z, S) \wedge \mathsf{monitoring}(exec, Z, Y, S)$
Should do
E9 $\mathsf{should\_do}(X, S) \leftarrow \mathsf{delegateChain}(exec, Y, X, S) \wedge \mathsf{provides}(X, S)$
E10 $\mathsf{should\_do}(X, S) \leftarrow \mathsf{requests}(X, S) \wedge \mathsf{provides}(X, S)$
Can satisfy
E11 $\mathsf{can\_satisfy}(X, S) \leftarrow \mathsf{should\_do}(X, S)$
E12 $\mathsf{can\_satisfy}(X, S) \leftarrow \mathsf{delegate}(exec, X, B, S) \wedge \mathsf{can\_satisfy}(B, S)$
E13 $\mathsf{can\_satisfy}(X, S) \leftarrow \mathsf{OR\_subgoal}(S_1, S) \wedge \mathsf{can\_satisfy}(X, S_1)$
E14 $\mathsf{can\_satisfy}(X, S) \leftarrow \mathsf{AND\_decomp}(S, S_1, S_2) \wedge \mathsf{can\_satisfy}(X, S_1) \wedge \mathsf{can\_satisfy}(X, S_2)$
Confident to can satisfy
E15 $\mathsf{confident}(satisfy, X, S) \leftarrow \mathsf{should\_do}(X, S)$
E16 $\mathsf{confident}(satisfy, X, S) \leftarrow \begin{cases} \mathsf{delegateChain}(exec, X, Y, S) \wedge \\ \mathsf{trustChain}(exec, X, Y, S) \wedge \mathsf{confident}(satisfy, Y, S) \end{cases}$
E17 $\mathsf{confident}(satisfy, X, S) \leftarrow \mathsf{OR\_subgoal}(S_1, S) \wedge \mathsf{confident}(satisfy, X, S_1)$
E18 $\mathsf{confident}(satisfy, X, S) \leftarrow \begin{cases} \mathsf{AND\_decomp}(S, S_1, S_2) \wedge \mathsf{confident}(satisfy, X, S_1) \\ \quad \wedge \mathsf{confident}(satisfy, X, S_2) \end{cases}$

Table 3. Axioms for execution

chains over monitoring steps. E8 and M4 have chains propagate to subgoals. According to E8 execution-trust flows top-down with respect to goal refinements. The axiom for monitoring M4 states that trustChain flows top-down with respect to goal refinements. M5 states that if an actor under monitoring delegates a service to another, then the monitor have to watch for the delegatee, that is, the monitor follows the delegation. M6 introduces the intensional predicate $\mathsf{confident}(mon, a, b, s)$: actor a is confident that there exists someone that monitors actor b for service s.

The remaining axioms describe how global properties of the model are defined. E9-10 state that an actor has to execute the service if he provides a service and if either some actor delegates the service to him, or he himself aims for the service. E11-12 state an actor, who requests for a service, can satisfy the service if either he provides it or he has delegated it to someone who can satisfy it. Goal refinements are taken care of by using the axioms E13-14. If an actor can satisfy at least one of the or-subgoals of a service, then he can satisfy the main service. Also, if he can satisfy all and-subgoals, then he can satisfy the main service.

The notion of confidence is captured by axioms E15-E18. An actor is confident that a service will be fulfilled, if he knows that all delegations have been done to trusted or monitored agents and that the agents who will ultimately execute the service, have the permission to do so. Goal refinements are taken care of by using axioms E17-18: if an actor is confident that at least one of the or-subgoals of a service will be fulfilled, then he can be confident that the service will be fulfilled. The axiom for and-decomposition is dual.

6.2 Formal Model for Permission

In Table 2 we also have predicates for modelling permission. The first set of predicates corresponds to the relations drawn by the requirements engineer. The predicate $\mathsf{owns}(a, s)$ holds if actor a owns service s. The owner of a service has full authority concerning access and usage of his services, and he can also delegate this authority to other actors. The intuition is that $\mathsf{delegate}(perm, a, b, s)$ holds if actor a at most delegates the permission to fulfill service s to actor b. The predicate $\mathsf{trust}(perm, a, b, s)$ holds is actor a trusts that actor b at most has the permission to fulfill service s. The predicate $\mathsf{monitoring}(perm, a, b, s)$ is the dual of the execution counterpart.

Also in this case other predicates are used to define interesting properties for the formal analysis by the requirement engineer. The predicates $\mathsf{delegateChain}(perm, a, b, s)$ and $\mathsf{trustChain}(perm, a, b, s)$ hold if there is a delegation, resp. a trust chain of permission among actor a and actor b. The basic idea of $\mathsf{has\_per}$ sums up the possible ways in which an actor can grab the permission on a service: either directly or by delegation. From the point of view of the owner, confidence means that the owner is confident that the permission that he has delegated will not be misused. Alternatively, the owner is confident that he has delegated permission only to trusted or monitored agents. This means that even if there is one untrusted or unmonitored delegation, then the owner could be uneasy about the likely misuse of his permissions. So, an owner is confident, if there is no likely misuse of his permission. It can be seen that there is an intrinsic double negation in the statement. So we try to model it using a predicate $\mathsf{diffident}(a, s)$. At any point of delegation of permission, the delegating agent is diffident, if the delegation is being done to an agent who is neither trusted not monitored or if the delegatee could be diffident himself. In this way, $\mathsf{confident}(owner, a, s)$ holds if the owner a is confident to give the permission on service s only to trusted actors.

Table 4 presents the axioms for modelling permission. P1 and P2 build a delegation chain of permission. P3-6 define the intensional versions, $\mathsf{trustChain}$ and $\mathsf{distrustChain}$ of the extensional predicates trust and $\mathsf{distrust}$ that are used to build (dis)trust chains by propagating (dis)trust of permission relations.

Delegation
P1 $\mathsf{delegateChain}(perm, X, Y, S) \leftarrow \mathsf{delegate}(perm, X, Y, S)$
P2 $\mathsf{delegateChain}(perm, X, Z, S) \leftarrow \mathsf{delegate}(perm, X, Y, S) \wedge \mathsf{delegateChain}(perm, Y, Z, S)$
Trust
P3 $\mathsf{distrustChain}(perm, X, Y, S) \leftarrow \mathsf{distrust}(perm, X, Y, S)$
P4 $\mathsf{distrustChain}(perm, X, Z, S) \leftarrow \begin{cases} \mathsf{trustChain}(perm, X, Y, S) \wedge \mathsf{distrust}(perm, Y, Z, S) \wedge \\ \quad \mathsf{not\ distrustChain}(perm, X, Y, S) \end{cases}$
P5 $\mathsf{trustChain}(perm, X, Y, S) \leftarrow \mathsf{trust}(perm, X, Y, S) \wedge \mathsf{not\ distrustChain}(perm, A, B, S)$
P6 $\mathsf{trustChain}(perm, X, Z, S) \leftarrow \begin{cases} \mathsf{trustChain}(perm, X, Y, S) \wedge \mathsf{trustChain}(perm, Y, Z, S) \wedge \\ \quad \mathsf{not\ distrustChain}(perm, X, Z, S) \end{cases}$
P7 $\mathsf{trustChain}(perm, X, Z, S) \leftarrow \mathsf{trustChain}(mon, X, Y, S) \wedge \mathsf{monitoring}(perm, Y, Z, S)$
P8 $\mathsf{trustChain}(perm, X, Y, S) \leftarrow \mathsf{subgoal}(S, S_1) \wedge \mathsf{trustChain}(perm, X, Y, S_1)$
M7 $\mathsf{trustChain}(mon, X, Z, S) \leftarrow \mathsf{trustChain}(perm, X, Y, S) \wedge \mathsf{trustChain}(mon, Y, Z, S)$
Monitoring
M8 $\mathsf{monitoring}(perm, Z, Y, S_1) \leftarrow \begin{cases} \mathsf{delegateChain}(perm, X, Y, S_1) \wedge \\ \mathsf{monitoring}(perm, Z, X, S) \wedge \mathsf{subgoal}(S_1, S) \end{cases}$
M9 $\mathsf{confident}(mon, X, Y, S) \leftarrow \mathsf{trust}(mon, X, Z, S) \wedge \mathsf{monitoring}(perm, Z, Y, S)$
Has permission
P9 $\mathsf{has\_per}(X, S) \leftarrow \mathsf{owns}(X, S)$
P10 $\mathsf{has\_per}(X, S) \leftarrow \mathsf{delegateChain}(perm, Y, X, S) \wedge \mathsf{has\_per}(Y, S)$
P11 $\mathsf{has\_per}(X, S_1) \leftarrow \mathsf{subgoal}(S_1, S) \wedge \mathsf{has\_per}(X, S)$
Owner is confident to give the service to trusted actors
P12 $\mathsf{confident}(owner, X, S) \leftarrow \mathsf{owns}(X, S) \wedge \mathsf{not\ diffident}(X, S)$
P13 $\mathsf{diffident}(X, S) \leftarrow \mathsf{delegateChain}(perm, X, Y, S) \wedge \mathsf{diffident}(Y, S)$
P14 $\mathsf{diffident}(X, S) \leftarrow \mathsf{delegateChain}(perm, X, Y, S) \wedge \mathsf{not\ trustChain}(perm, X, Y, S)$
P15 $\mathsf{diffident}(X, S) \leftarrow \mathsf{subgoal}(S_1, S) \wedge \mathsf{diffident}(X, S_1)$

Table 4. Axioms for permission

P5 and P6 build a trust chain for permission; P7 builds chains over monitoring steps. P8 has the chain propagate through subgoals. If an actor trusts that another will not overstep the set of actions required to fulfill a subgoal of a service, then the first can trust the last not to overstep the set of actions required to fulfill the service. The permission trust, with respect to goal refinements, flows bottom-up. M7 is used to build a trust chain for monitor. M8 states that if an actor under monitoring delegates a service to another, then the monitor have to watch for the delegatee, that is, the monitor follows the delegation. M9 is the permission counterpart of M6. The owner of a service has full authority concerning access and disposition of it. Thus, P9 states that if an actor owns a service, he has permission on it. P10 states that the delegatee has permission on the service. P11 propagates permission through subgoals. The notion of confidence and diffidence that we have sketched above is captured by the axioms P12-P16.

6.3 Combining Execution and Permission

More sophisticated properties require reasoning with both execution and permission. To this end, we introduce some notions that put together these two notions. In Table

Can see the service fulfilled (can execute)	
Ax1	$\mathsf{can\_execute}(X,S) \leftarrow \mathsf{should\_do}(X,S) \wedge \mathsf{has\_per}(X,S)$
Ax2	$\mathsf{can\_execute}(X,S) \leftarrow \mathsf{delegateChain}(exec,X,Y,S) \wedge \mathsf{can\_execute}(Y,S)$
Ax3	$\mathsf{can\_execute}(X,S) \leftarrow \mathsf{OR\_subgoal}(S_1,S) \wedge \mathsf{can\_execute}(X,S_1)$
Ax4	$\mathsf{can\_execute}(X,S) \leftarrow \begin{cases} \mathsf{AND\_decomp}(S,S_1,S_2) \wedge \mathsf{can\_execute}(X,S_1) \\ \wedge\ \mathsf{can\_execute}(X,S_2) \end{cases}$
Confident to see the service fulfilled (confident to execute)	
Ax5	$\mathsf{confident}(exec,X,S) \leftarrow \mathsf{should\_do}(X,S) \wedge \mathsf{has\_per}(X,S)$
Ax6	$\mathsf{confident}(exec,X,S) \leftarrow \begin{cases} \mathsf{delegateChain}(exec,X,Y,S) \wedge \\ \mathsf{trustChain}(exec,X,Y,S) \wedge \mathsf{confident}(exec,Y,S) \end{cases}$
Ax7	$\mathsf{confident}(exec,X,S) \leftarrow \mathsf{OR\_subgoal}(S_1,S) \wedge \mathsf{confident}(exec,X,S_1)$
Ax8	$\mathsf{confident}(exec,X,S) \leftarrow \begin{cases} \mathsf{AND\_decomp}(S,S_1,S_2) \wedge \mathsf{confident}(exec,X,S_1) \\ \wedge\ \mathsf{confident}(exec,X,S_2) \end{cases}$
Need to know	
Ax9	$\mathsf{need\_to\_have\_perm}(X,S) \leftarrow \mathsf{should\_do}(X,S)$
Ax10	$\mathsf{need\_to\_have\_perm}(X,S) \leftarrow \begin{cases} \mathsf{delegate}(perm,X,Y,S) \wedge \mathsf{need\_to\_have\_perm}(Y,S) \\ \wedge\ \mathsf{not\ other\_delegater}(X,Y,S) \end{cases}$
Ax11	$\mathsf{other\_delegater}(X,Y,S) \leftarrow \begin{cases} \mathsf{delegate}(perm,Z,Y,S) \wedge \\ \mathsf{need\_to\_have\_perm}(Y,S) \wedge X \neq Z \end{cases}$

Table 5. Axioms Involving both permission and execution

5 we present the notions from both the point of view of the requester and the point of view of the owner. The predicate $\mathsf{can\_execute}(a,s)$ holds if actor a can see service s fulfilled. The predicate $\mathsf{confident}(exec,a,s)$ holds if actor a is confident to see service s fulfilled. Actor a, who aims for service s, is confident that s will be fulfilled, if he knows that all delegations have been done to trusted or monitored agents and that the agents who will ultimately execute the service, have the permission to do so. This is done using the axioms Ax5-6. Goal refinements are taken care of by using the axioms Ax7-8. If a is confident that at least one of the or-subgoals of s will be fulfilled, then a can be confident that s will be fulfilled. Also, if a is confident that all and-subgoals of s will be fulfilled, then a can be confident that s will be fulfilled.

Owners may wish to delegate permissions to providers only if the latter actually do need the permission. The last part of Table 5 defines the predicates that are necessary to analyze *need-to-know* properties. As a result of absence of diffidence, the owner can be confident that his permission will not be misused. But has this permission reached the agents who actually *need* it? The owner might also want to ensure that there has been not unwanted delegation of permission. This can be achieved by identifying the agents who actually *need-to-know* (or rather *need-to-have*) the permission. This set of axioms captures also the possibility of having alternate paths of permission delegations. In this case the formal analysis will not yield one model but multiple models in which only one path of delegation is labeled by the need-to-have property and the others are not.

Example 14 (Figure 9). Alice and Carol (7 and 8) have both received the consent (permission) by Bob (1) for using his personal data, and both delegate it to the faculty

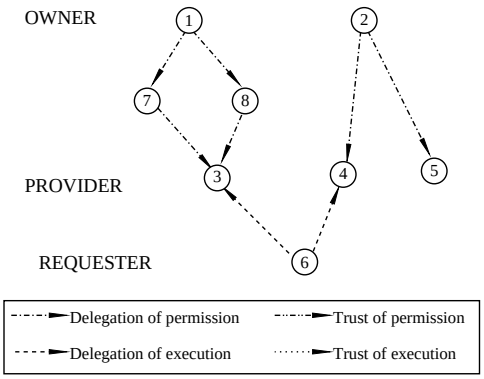


Fig. 9. Need-to-Know and Multiple Permissions Paths

From Tropos to Secure Tropos
ST1 $\mathsf{trust}(exec, X, Y, G) \leftarrow \mathsf{depends}(X, Y, G)$
ST2 $\mathsf{delegate}(exec, X, Y, G) \leftarrow \mathsf{depends}(X, Y, G)$
ST3 $\mathsf{trust}(perm, Y, X, R) \leftarrow \mathsf{depends}(X, Y, R)$
ST4 $\mathsf{delegate}(perm, Y, X, R) \leftarrow \mathsf{depends}(X, Y, R)$
From Secure Tropos to Tropos
ST5 $\mathsf{depends}(X, Y, G) \leftarrow \mathsf{trust}(exec, X, Y, G) \wedge \mathsf{delegate}(exec, X, Y, G) \wedge$ **not** $\mathsf{distrust}(exec, X, Y, G)$
ST6 $\mathsf{depends}(X, Y, R) \leftarrow \mathsf{trust}(perm, Y, X, R) \wedge \mathsf{delegate}(perm, Y, X, R) \wedge$ **not** $\mathsf{distrust}(perm, Y, X, R)$

Table 6. Axioms for mapping Tropos into Secure Tropos and vive versa

secretariat (3), which must have the permission to provide the data to Paul (6), the university tutor who should provide personal counseling to Bob. In this case only one of either Alice or Carol needs to have the permission.

6.4 Other features

In Table 6 there are the axioms to map Tropos dependency into Secure Tropos framework and vice versa. Notice that ST1-2 and ST5 have also to be repeated for the case where the dependum is a plan.

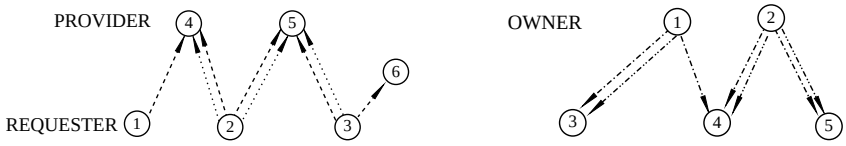
Table 7 presents the axioms for role hierarchy and for mapping relations from social level to individual level. The predicate specialize is the intensional version of is_a, whereas instance is intensional version of play. Axioms SI1-13 have to be repeated replacing the predicate instance with specialize and predicate agent with role for completing social level with respect to role hierarchy.

6.5 Analysis and Verification

Design *properties* are not enforced with axioms for two reasons. At first the actual system drawn by the requirement engineer may not satisfy them, and therefore the missing

Role Hierarchy
RH1 $\mathsf{specialize}(T, Q) \leftarrow \mathsf{is\_a}(T, Q)$
RH2 $\mathsf{specialize}(T, Q) \leftarrow \mathsf{specialize}(T, V) \wedge \mathsf{is\_a}(V, Q)$
RH3 $\mathsf{instance}(A, T) \leftarrow \mathsf{play}(A, T)$
RH4 $\mathsf{instance}(A, T) \leftarrow \mathsf{instance}(A, Q) \wedge \mathsf{specialize}(Q, T)$
From social level to individual level
SI1 $\mathsf{provides}(A, S) \leftarrow \mathsf{provides}(T, S) \wedge \mathsf{instance}(A, T)$
SI2 $\mathsf{requests}(A, S) \leftarrow \mathsf{requests}(T, S) \wedge \mathsf{instance}(A, T)$
SI3 $\mathsf{owns}(A, S) \leftarrow \mathsf{owns}(T, S) \wedge \mathsf{instance}(A, T)$
SI4 $\mathsf{trust}(exec, A, B, S) \leftarrow \mathsf{trust}(exec, T, Q, S) \wedge \mathsf{instance}(A, T) \wedge \mathsf{instance}(B, Q)$
SI5 $\mathsf{trust}(perm, A, B, S) \leftarrow \mathsf{trust}(perm, T, Q, S) \wedge \mathsf{instance}(A, T) \wedge \mathsf{instance}(B, Q)$
SI6 $\mathsf{distrust}(exec, A, B, S) \leftarrow \mathsf{distrust}(exec, T, Q, S) \wedge \mathsf{instance}(A, T) \wedge \mathsf{instance}(B, Q)$
SI7 $\mathsf{distrust}(perm, A, B, S) \leftarrow \mathsf{distrust}(perm, T, Q, S) \wedge \mathsf{instance}(A, T) \wedge \mathsf{instance}(B, Q)$
SI8 $\mathsf{delegate}(exec, A, B, S) \leftarrow \mathsf{delegate}(exec, T, Q, S) \wedge \mathsf{instance}(A, T) \wedge \mathsf{instance}(B, Q)$
SI9 $\mathsf{delegate}(perm, A, B, S) \leftarrow \mathsf{delegate}(perm, T, Q, S) \wedge \mathsf{instance}(A, T) \wedge \mathsf{instance}(B, Q)$
SI10 $\mathsf{monitoring}(exec, A, B, S) \leftarrow \mathsf{monitoring}(exec, T, Q, S) \wedge \mathsf{instance}(A, T) \wedge \mathsf{instance}(B, Q)$
SI11 $\mathsf{monitoring}(perm, A, B, S) \leftarrow \mathsf{monitoring}(perm, T, Q, S) \wedge \mathsf{instance}(A, T) \wedge \mathsf{instance}(B, Q)$
SI12 $\mathsf{trust}(mon, A, B, S) \leftarrow \mathsf{trust}(mon, T, Q, S) \wedge \mathsf{instance}(A, T) \wedge \mathsf{instance}(B, Q)$
SI13 $\mathsf{depends}(A, B, S) \leftarrow \mathsf{depends}(T, Q, S) \wedge \mathsf{instance}(A, T) \wedge \mathsf{instance}(B, Q)$

Table 7. Axioms for role hierarchy and for mapping social level into individual level



(a) From requesters' viewpoint (b) From owners' viewpoint

Fig. 10. Design for delegation of execution and permission

link may be actually a bug. Second, there might be many ways in which a requirement engineer may wish to fulfill desired properties. We use the DLV system[7] to verify security properties with respect to a Secure Tropos model.

In Table 8 we use the $A \Rightarrow? B$ to mean that one must check that each time A holds it is desirable that B also holds. In Datalog this can be represented as the constraint :- A, not B. If the set of features is not consistent, i.e., they cannot all be simultaneously satisfied, the system is inconsistent, and hence it is not secure. This also guarantee us that our proposed axioms are consistent if we check for consistency of the model without trying to enforce any property.

Pro1 states that if there is a delegation chain either the delegater trusts the delegatee or there is the monitor and the delegater trust the monitor. Pro2 states that a requester wants to can satisfy his goals, and Pro3 states that a requester wants to be confident to satisfy the service.

[7] http://www.dbai.tuwien.ac.at/proj/dlv

Execution	
Pro1	$\mathsf{delegateChain}(exec, X, Y, S) \Rightarrow? \mathsf{trustChain}(exec, X, Y, S)$
Pro2	$\mathsf{requests}(X, S) \Rightarrow? \mathsf{can\_satisfy}(X, S)$
Pro3	$\mathsf{requests}(X, S) \Rightarrow? \mathsf{confident}(satisfy, X, S)$
Pro4	$\mathsf{should\_do}(X, S) \Rightarrow? \mathsf{not\ delegateChain}(exec, X, Y, S)$
Permission	
Pro5	$\mathsf{delegateChain}(perm, X, Y, S) \Rightarrow? \mathsf{trustChain}(perm, X, Y, S)$
Pro6	$\mathsf{owns}(X, S) \Rightarrow? \mathsf{confident}(owner, X, S)$
Pro7	$\mathsf{owns}(X, S) \Rightarrow? \mathsf{not\ delegateChain}(perm, Y, X, S) \wedge X \neq Y$
Execution & Permission	
Pro8	$\mathsf{requests}(X, S) \Rightarrow? \mathsf{can\_execute}(X, S)$
Pro9	$\mathsf{requests}(X, S) \Rightarrow? \mathsf{confident}(exec, X, S)$
Pro10	$\mathsf{owns}(X, S) \Rightarrow? \mathsf{need\_to\_have\_perm}(X, S)$
Pro11	$\mathsf{owns}(X, S) \Rightarrow? \mathsf{need\_to\_have\_perm}(X, S) \wedge \mathsf{confident}(owner, X, S)$

Table 8. Desirable Properties of a Design

Example 15 (Figure 10(a)). Bob and Bert (1 and 2) need counseling. They can receive it (formal relation can satisfy) because they delegate the execution to Paul and Peter (4 and 5), while Bill (3) cannot receives all necessary advices because he requested some of them only to Alice (6) which is not able to provide counseling on faculty matters.

Bob is also confident to receive all counseling he needs since he delegates the execution to Paul and Peter (4 and 5) whom he trusts, while Bert is not confident since he delegates to Paul (4) that he does not trust.

Pro4 states that if an actor provides a service, then, if either some actor delegates the service to him, or if he himself requests the service, then he has to execute the service without further delegation. Pro5 states that if there is a delegation chain, either the delegater trusts the delegatee or there is the monitor. Pro6 states that the owner of the service has to be confident to give the service to trusted actors, and Pro7 states that a service cannot come back to the owner.

Example 16 (Figure 10(b)). Bob and Bert (1 and 2) need to provide their personal data for receiving accurate counseling. Bob is confident on his personal data since he delegates the permission on it to two Paul and Peter (4 and 5) who he trusts to use the data at most for counseling. On the other hand, Bert is not confident on her data since she delegates it to Paul (4) whom she does not trust to keep her information confidential.

This example is very close to the example that we have previously seen on misplaced delegation (Example 15). What changes is what can be obtained by poor Bert. In the former case he is afraid to receive a bad advice (delegation of execution), in the latter that her information can be used for other things than providing counseling.

The last part of Table 8 shows properties to verify at-most model and at-least model at the same time. Pro8 states that the requester has to can see the service fulfilled. Pro9 states that the requester has to be confident to see the service fulfilled.

Table 9 presents the properties used to identifying conflicts that occur when both a trust and a distrust relations exist among two actors for the same service. Pro1-2 are

TC1 $\mathsf{trustChain}(exec, X, Y, S) \Rightarrow ?\mathsf{not\ distrustChain}(exec, X, Y, S)$
TC2 $\mathsf{trustChain}(perm, X, Y, S) \Rightarrow ?\mathsf{not\ distrustChain}(perm, X, Y, S)$
TC3 $\mathsf{trustChain}(exec, A, B, S) \Rightarrow ? \begin{cases} \mathsf{not\ distrustChain}(exec, T, Q, S) \wedge \\ \mathsf{instance}(A, T) \wedge \mathsf{instance}(B, Q) \end{cases}$
TC4 $\mathsf{trustChain}(perm, A, B, S) \Rightarrow ? \begin{cases} \mathsf{not\ distrustChain}(perm, T, Q, S) \wedge \\ \mathsf{instance}(A, T) \wedge \mathsf{instance}(B, Q) \end{cases}$
TC5 $\mathsf{distrustChain}(exec, A, B, S) \Rightarrow ? \begin{cases} \mathsf{not\ trustChain}(exec, T, Q, S) \wedge \\ \mathsf{instance}(A, T) \wedge \mathsf{instance}(B, Q) \end{cases}$
TC6 $\mathsf{distrustChain}(perm, A, B, S) \Rightarrow ? \begin{cases} \mathsf{not\ trustChain}(perm, T, Q, S) \wedge \\ \mathsf{instance}(A, T) \wedge \mathsf{instance}(B, Q) \end{cases}$

Table 9. Properties for identifying conflicts

C1 $\{\mathsf{monitoring}(exec, M, B, S)\} \leftarrow \begin{cases} \mathsf{distrustChain}(exec, A, B, S) \wedge \\ \mathsf{trustChain}(exec, T, Q, S) \wedge \\ \mathsf{instance}(A, T) \wedge \mathsf{instance}(B, Q) \wedge \\ \mathsf{trustChain}(mon, A, M, S) \end{cases}$
C2 $\{\mathsf{monitoring}(perm, M, B, S)\} \leftarrow \begin{cases} \mathsf{distrustChain}(perm, A, B, S) \wedge \\ \mathsf{trustChain}(perm, T, Q, S) \wedge \\ \mathsf{instance}(A, T) \wedge \mathsf{instance}(B, Q) \wedge \\ \mathsf{trustChain}(mon, A, M, S) \end{cases}$

Table 10. Axioms for solving conflicts

E4′ $\mathsf{distrust}(exec, A, B, S) \leftarrow \begin{cases} \mathsf{distrust}(exec, T, Q, S) \wedge \mathsf{instance}(A, T) \wedge \\ \mathsf{instance}(B, Q) \wedge \mathbf{not}\ \mathsf{confident}(mon, A, B, S) \end{cases}$
P4′ $\mathsf{distrust}(perm, A, B, S) \leftarrow \begin{cases} \mathsf{distrust}(perm, T, Q, S) \wedge \mathsf{instance}(A, T) \wedge \\ \mathsf{instance}(B, Q) \wedge \mathbf{not}\ \mathsf{confident}(mon, A, B, S) \end{cases}$

Table 11. Axioms in order to support monitoring

used to identify generic conflicts and correspond to Definition 1 and 2. These properties apply to both social level and individual level, independently and so A and B have to be typed as role for the social level and as agents for the individual level. Pro1-2 can be refined in order to identify conflicts of the form of Fig. 8(c) (Pro3-4) and Fig. 8(b) (Pro5-6).

Table 10 formalizes the proposal for solving conflicts when there is a trust relation at social level and a distrust relation at individual level. In order to accommodate C1-2 in our framework we have to modify axioms Ax6-7 in Table 7. The new version of these axioms is given in Table 11.

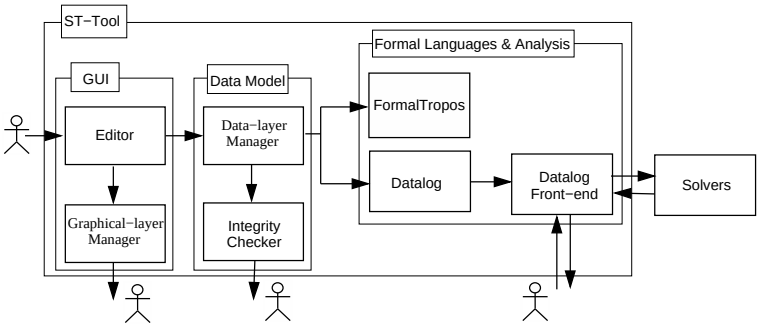


Fig. 11. The Architecture Overview

7 Computer Aided SRE

ST-Tool [24, 29] is a CASE tool for design and verification of functional and security requirements, and has been designed to support the Secure Tropos methodology. It provides a user interface for drawing Secure Tropos models, support for translating automatically graphical models into formal specifications and a front-end with external tools for model checking.

ST-Tool is mainly composed of two parts: the ST-Tool kernel and external solvers. ST-Tool kernel has an architecture comprised of three major parts, each of which is comprised of modules. Next, we will discuss these modules and their interconnections. In Fig. 11, the modules of ST-Tool are shown, their interrelations are also indicated.

The tool provides a graphical user interface (GUI), through which system designers can manage all the components and functionalities of the tool. A screenshot of the interface is shown in Fig. 12. To manage visual editing features and data management consistency at the same time, we have adopted a two-layer solution: a graphical layer and a data layer. In graphical layer, models are shown as graphs where actors and services are nodes, and relations are arcs. Each visual object refers to a data object. The collection of data objects is the data layer. The GUI's key component is the *Editor Module*. This module allows the user to visually insert, edit or remove graphical objects in the graphical layer and object properties in the data layer. A second GUI component is the *Graphical-layer Manager (GM) Module* that manages graphical objects and their visualization. It supports goal refinement by associating a goal diagram with each actor and then allows to collapse actors and services in order to maintain readable diagrams. Further, GM permits to display one or more views of a diagram at the same time, namely dependency model (aka Tropos model), delegation models, and trust models.

The *Data-layer Manager (DM) Module* is responsible for building and maintaining data corresponding to graphical objects. For example, DM manages misalignments between social relations and their graphical representation. Actually, GM uses arcs to connect two nodes to each other, while many Secure Tropos relations are ternary. DM rebuilds these relations by linking two appropriate graphical objects (the two arcs) to the same data object (the relation). ST-Tool allows users to save models through the DM module that stores a neutral description of the entire model in `.xml` format files. A support for detecting errors and warnings during the design phase is provided by

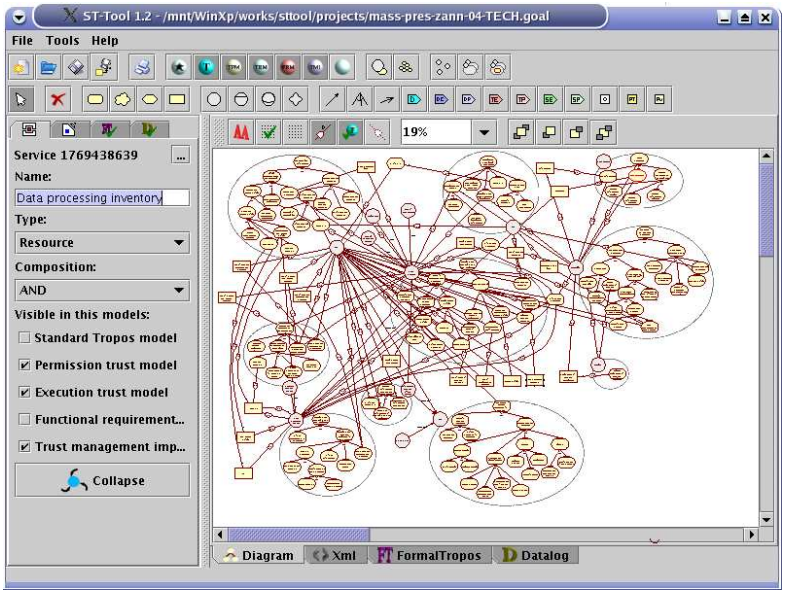


Fig. 12. ST-Tool screenshot

the *Integrity Checker Module*. This module analyzes models stored in the DM module and reports errors such as "orphan relations" (i.e. relations where an arc is missing) and "isolated nodes" (i.e. services not involved in any relations). Warnings are different from errors: they are failure of integrity constraints, like errors, but the designer may be perfectly happy with a design that does not satisfy them. Integrity Checker reports warnings, for example, when more than one service have the same name.[8]

After drawing so many nice diagrams, system designers may want to check whether the models derived so far satisfy some general desirable properties. To support formal analysis, ST-Tool allows automatic transformations from the `.xml` file stored by DM into Formal Tropos [20] and Datalog specifications. These transformations are performed, respectively, by two different modules: *Formal Tropos Module* and *Datalog Module*. The resulting specifications are displayed by selecting the corresponding panel.

The process for completing and checking models is controlled by the *Datalog Front-end (DF) Module*. Through this module, requirement engineers can choose the axioms to complete the model and the properties to be verified on it. Properties are grouped into Authorization, Availability, Integrity and Need-to-know categories, so that engineers only need to specify the categories they wants to verify to include the corresponding rule set. Once designers are confident with the model, the resulting Datalog specification is given in input to some external solvers that verify the consistency of the model corresponding to the specifications. Then, the solver output is parsed by the DF module in order to present in a more user-readable format. A scheme of the entire process for modelling and analyzing security requirements is given in Fig. 13.

[8] More than one service with the same name are needed to represent delegation and trust chains.

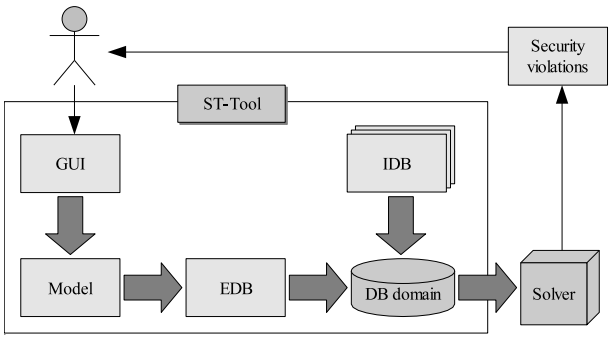


Fig. 13. ST-Tool: the analysis cycle

Solver	cmodels-1			cmodels-2			smodels			assat			dlv		
N. Ins.	R	Wall	CPU	R	Wall	CPU	R	Wall	CPU	R	Wall	CPU	R	Wall	CPU
0	0	0m13.32s	0m0.25s	0	0m13.53s	0m0.13s	0	0m14.83s	0m0.13s	0	0m14.82s	0m0.13s	0	0m0.12s	0m0.01s
24	0	0m59.08s	0m0.61s	0	0m58.99s	0m0.57s	0	1m5.15s	0m0.56s	0	1m4.92s	0m0.59s	0	0m0.31s	0m0.00s
45	0	2m33.69s	0m2.06s	0	2m33.77s	0m1.73s	0	2m50.51s	0m1.68s	0	2m50.18s	0m1.75s	0	0m0.67s	0m0.02s
62	1	0m41.19s	0m1.80s	1	0m41.04s	0m1.74s	1	0m46.28s	0m1.66s	1	0m46.72s	0m1.60s	0	0m0.95s	0m0.01s
113	1	0m47.94s	0m1.72s	1	0m47.70s	0m1.76s	1	0m54.34s	0m1.63s	1	0m54.27s	0m1.71s	0	0m2.42s	0m0.02s
166	1	0m27.73s	0m1.58s	1	0m27.77s	0m1.55s	1	0m32.71s	0m1.75s	1	0m33.74s	0m1.86s	0	0m5.05s	0m0.08s

Table 12. Experimental Result

We use different ASP solvers for the requirements analysis, namely ASSAT,[9] Cmodels,[10] Smodels,[11] and DLV.[12] ASSAT, Cmodels, and Smodels work with grounded logic programs generated by Lparse [54]. In particular, Cmodels and ASSAT use SAT solvers as research engine for determining the solution, while Smodels uses general-purpose answer set solvers. Finally, DLV is developed as a deductive database system.

In order to compare the different solvers, we have tested them on a pool of benchmarks based on a comprehensive case study on the compliance to the Italian security and privacy legislation of public administrations such as universities, local governments and health care authorities [43]. Benchmarks are defined from the structure of the organization (base case) by adding a growing number of agents (instances) playing the roles occurring in the model.

The benchmarks evaluation results of the experiments carried out are reported in Table 12. The experiments were executed on a bi-processor XEON, 3.2 GHz, 1 MB of Chache, 4GB of RAM, running Linux. For each problem we report the time used to complete the analysis (Wall) and by CPU. However, Wall and CPU reported in Table 12 do not take into account the time spent by Lparse that Cmodels, Smodels and Assat use for grounding. Further, with "0" we mark the experiments that complete successfully, while with "1" we mark those experiments that fail for some reason such as memory limits exceeded. The experiments show that DLV system is more efficient than the other

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

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

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

[12] http://www.dbai.tuwien.ac.at/proj/dlv/

solvers. Further, Cmodels, Smodels and ASSAT are not able to find a solution after a certain number of instances since Lparse exceeds memory limit.

8 Conclusions

Security Requirements Engineering is one of the challenging field for computer security research. Here we have sketched the overall methodological issues that underpins the design of a novel methodology for security design.

Looking back at our proposed classification, this work is well placed within the meta-level modelling 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 Tropos/i* original methodology.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic Databases. In *Proc. of VLDB'02*, pages 143–154. Morgan Kaufmann, 2002.
3. R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. An Implementation of P3P Using Database Technology. In *Proc. of EDBT'04*, *LNCS 2992*, pages 845–847. Springer-Verlag, 2004.
4. R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Computer Publishing, 2001.
5. A. I. Antòn and J. B. Earp. A requirements taxonomy for reducing Web site privacy vulnerabilities. *Requirements Eng.*, 9(3):169–185, 2004.
6. A. I. Antòn, J. B. Earp, and A. Reese. Analyzing Website privacy requirements using a privacy goal taxonomy. In *Proc. of RE'02*, pages 23–31. IEEE Press, 2002.
7. T. Aura. On the Structure of Delegation Networks. In *Proc. of 1998 CSFW*, pages 14–26. IEEE Press, 1998.
8. M. Backes, G. Karjoth, W. Bagga, and M. Schunter. Efficient comparison of enterprise privacy policies. In *Proc. of SAC'04*, 2004.
9. M. Backes, B. Pfitzmann, and M. Schunter. A Toolkit for Managing Enterprise Privacy Policies. In *Proc. of ESORICS'03*, *LNCS 2808*, pages 162–180. Springer-Verlag, 2003.
10. P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. TROPOS: An Agent-Oriented Software Development Methodology. *JAAMAS*, 8(3):203–236, 2004.
11. C. Castelfranchi and R. Falcone. Principles of trust for MAS: Cognitive anatomy, social importance and quantification. In *Proc. of ICMAS'98*, pages 72–79. IEEE Press, 1998.
12. L. K. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Publishing, 2000.
13. L. Cranor, M. Langheinrich, M. Marchiori, and J. Reagle. The Platform for Privacy Preferences 1.0 (P3P1.0) Specification. W3C Recommendation, Apr. 2002.
14. R. Crook, D. Ince, L. Lin, and B. Nuseibeh. Security Requirements Engineering: When Anti-requirements Hit the Fan. In *Proc. of RE'02*, pages 203–205. IEEE Press, 2002.
15. J. DeTreville. Binder, a logic-based security language. In *Proc. of 2002 IEEE Symp. on Sec. and Privacy*, pages 95–103. IEEE Press, 2002.
16. P. T. Devanbu and S. G. Stubblebine. Software engineering for security: a roadmap. In *Proc. of ICSE'00*, pages 227–239, 2000.
17. T. Doan, S. Demurjian, T. C. Ting, and A. Ketterl. MAC and UML for secure software design. In *Proc. of FMSE'04*, pages 75–85. ACM Press, 2004.

18. D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed nist standard for role-based access control. *TISSEC*, 4(3):224–274, 2001.
19. R. Fredriksen, M. Kristiansenand, B. A. G. K. Stølen, T. A. Opperud, and T. Dimitrakos. The CORAS framework for a model-based risk management process. In *Proc. of SAFECOMP'02*, *LNCS 2434*, pages 94–105, 2002.
20. A. Fuxman, L. Liu, M. Pistore, M. Roveri, and J. Mylopoulos. Specifying and analyzing early requirements: Some experimental results. In *Proc. of RE'03*. IEEE Press, 2003.
21. G. Gans, M. Jarke, S. Kethers, and G. Lakemeyer. Modeling the Impact of Trust and Distrust in Agent Networks. In *Proc. of AOIS'01*, pages 45–58, 2001.
22. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of the 5th Int. Conf. on Log. Prog.*, pages 1070–1080. MIT Press, 1988.
23. 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*, *LNCS 2813*, pages 263–276. Springer-Verlag, 2003.
24. P. Giorgini, F. Massacci, J. Mylopoulos, A. Siena, and N. Zannone. ST-Tool: A CASE Tool for Modeling and Analyzing Trust Requirements. In *Proc. of iTrust'05*, *LNCS 3477*, pages 415–419. Springer-Verlag, 2005.
25. P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone. Filling the gap between Requirements Engineering and Public Key/Trust Management Infrastructures. In *Proc. of EuroPKI'04*, *LNCS 3093*, pages 98–111. Springer-Verlag, 2004.
26. P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone. Requirements Engineering meets Trust Management: Model, Methodology, and Reasoning. In *Proc. of iTrust'04*, *LNCS 2995*, pages 176–190. Springer-Verlag, 2004.
27. P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone. Modeling Security Requirements Through Ownership, Permission and Delegation. In *Proc. of RE'05*, 2005. To appear.
28. P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone. Modelling Social and Individual Trust in Requirements Engineering Methodologies. In *Proc. of iTrust'05*, *LNCS 3477*, pages 161–176. Springer-Verlag, 2005.
29. P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone. ST-Tool: A CASE Tool for Security Requirements Engineering. In *Proc. of RE'05*, 2005. To appear.
30. Z. Guessoum, M. Ziane, and N. Faci. Monitoring and Organizational-Level Adaptation of Multi-Agent Systems. In *Proc. of AAMAS'04*, pages 514–521. ACM Press, 2004.
31. M. Hannoun, J. S. Sichman, O. Boissier, and C. Sayettat. Dependence Relations between Roles in a Multi-Agent System: Towards the Detection of Inconsistencies in Organization. In *Proc. of MABS'98*, *LNCS 1534*, pages 169–182. Springer-Verlag, 1998.
32. Q. He and A. I. Antón. A Framework for Modeling Privacy Requirements in Role Engineering. In *Proc. of the 9th Int. Workshop on Requirements Eng. : Found. for Software Quality*, pages 137–146, 2003.
33. T. Jaeger and A. Prakash. Requirements of role-based access control for collaborative systems. In *Proc. of 1st ACM Workshop on Role-Based Access Control*, pages 53–64. ACM Press, 1995.
34. A. J. I. Jones and M. J. Sergot. A Formal Characterisation of Institutionalised Power. *J. of the Interest Group in Pure and Appl. Log.*, 4(3):429–445, 1996.
35. J. Jürjens. *Secure Systems Development with UML*. Springer-Verlag, 2004.
36. G. A. Kaminka, D. V. Pynadath, and M. Tambe. Monitoring Teams by Overhearing: A Multi-Agent Plan-Recognition Approach. *JAIR*, 17:83–135, 2002.
37. G. Karjoth, M. Schunter, and M. Waidner. Platform for Enterprise Privacy Practices: Privacy-enabled Management of Customer Data. In *Proc. of PET'02*. Springer-Verlag, 2002.
38. N. Li, B. N. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *TISSEC*, 6(1):128–171, 2003.

39. N. Li, J. C. Mitchell, and W. H. Winsborough. Design of A Role-based Trust-management Framework. In *Proc. of 2002 IEEE Symp. on Sec. and Privacy*, pages 114–130. IEEE Press, 2002.
40. L.-C. Lin, B. Nuseibeh, D. Ince, M. Jackson, and J. Moffett. Analysing Security Threats and Vulnerabilities Using Abuse Frames. Technical Report 2003/10, The Open University, 2003.
41. L. Liu, E. S. K. Yu, and J. Mylopoulos. Security and Privacy Requirements Analysis within a Social Setting. In *Proc. of RE'03*, pages 151–161. IEEE Press, 2003.
42. T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *Proc. of UML'02*, *LNCS 2460*, pages 426–441. Springer-Verlag, 2002.
43. F. Massacci, M. Prest, and N. Zannone. Using a Security Requirements Engineering Methodology in Practice: The compliance with the Italian Data Protection Legislation. *Comp. Standards & Interfaces*, 27(5):445–455, 2005. An extended version is available as Technical report DIT-04-103 at `eprints.biblio.unitn.it`.
44. J. McDermott and C. Fox. Using Abuse Case Models for Security Requirements Analysis. In *Proc. of ACSAC'99*, pages 55–66. IEEE Press, 1999.
45. H. Mouratidis, P. Giorgini, and G. Manson. Modelling secure multiagent systems. In *Proc. of AAMAS'03*, pages 859–866. ACM Press, 2003.
46. H. Nwana. Software agents: An overview. *Knowledge Engineering Review J.* , 11(3), 1996.
47. S. Osborn, R. Sandhu, and Q. Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *TISSEC*, 3(2):85–106, 2000.
48. L. Ponemon. What Keeps Security Professionals Up At Night?, April 2003. URL: http://www.darwinmag.com/read/040103/threats.html.
49. I. Ray, N. Li, R. France, and D.-K. Kim. Using UML to visualize role-based access control constraints. In *Proc. of SACMAT'04*, pages 115–124. ACM Press, 2004.
50. P. Samarati and S. D. C. di Vimercati. Access Control: Policies, Models, and Mechanisms. In *FOSAD 2001/2002*, *LNCS 2946*, pages 137–196. Springer-Verlag, 2001.
51. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Comp.*, 29(2):38–47, 1996.
52. G. Sindre and A. L. Opdahl. Eliciting security requirements with misuse cases. *Requirements Eng.*, 10(1):34–44, 2005.
53. W. Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice-Hall, Englewood Cliffs, New Jersey, 1999.
54. T. Syrjänen. *Lparse 1.0: User's Manual*. Helsinki University of Technology, 2000.
55. A. Toval, A. Olmos, and M. Piattini. Legal requirements reuse: a critical success factor for requirements quality and personal data protection. In *Proc. of RE'02*, pages 95 –103. IEEE Press, 2002.
56. T. Tryfonas, E. Kiountouzis, and A. Poulymenakou. Embedding security practices in contemporary information systems development approaches. *Inform. Management and Comp. Sec.*, 9:183–197, 2001.
57. A. van Gelder. The alternating fixpoint of logic programs with negation. In *Proc. of PODS'89*, pages 1–10. ACM Press, 1989.
58. A. van Lamsweerde, S. Brohez, R. De Landtsheer, and D. Janssens. From System Goals to Intruder Anti-Goals: Attack Generation and Resolution for Security Requirements Engineering. In *Proc. of RHAS'03*, pages 49–56, 2003.
59. A. van Lamsweerde and E. Letier. Handling Obstacles in Goal-Oriented Requirements Engineering. *TSE*, 26(10):978–1005, 2000.
60. J. Viega and G. McGraw. *Building Secure Software*. Addison-Wesley, 2001.
61. E. S. K. Yu. Agent-Oriented Modelling: Software versus the World. In *Proc. of AOSE'01*, *LNCS 2222*, pages 206–225. Springer-Verlag, 2001.

62. P. Zave. Classification of research efforts in requirements engineering. *CSUR*, 29(4):315–321, 1997.