Tropos: Security

Paolo Giorgini
Department of Information and Communication Technology
University of Trento - Italy
http://www.dit.unitn.it/~pgiorgio

Agent-Oriented Software Engineering course
Laurea Specialistica in Informatica
A.A. 2007-2008
Talk Outline

- Security Requirements Engineering
- Reviewing the state-of-the-art
- Secure Tropos
- Modeling Framework
  - Formal Framework
  - Computer-Aided SRE
  - Ongoing and Future Work
- Demo and Case Study
Security and Requirements Engineering

- Traditional RE approaches treat security as a non-functional requirement
- According to this view
  - security requirements are modeled as quality constraints under which the system must operate
- These need to be integrated with other non-functional requirements (e.g., reliability and performance), to be dealt with by the software development process
Security in SE Practice

- The usual approach towards the inclusion of security within a system is to identify security requirements after system design.
- Most SE proposals focus on protection aspects of security and explicitly deal with a series of security services (integrity, availability, etc.) and related protection mechanisms (password, crypto, etc.).
- Security mechanisms have to be fitted into a pre-existing design, which may not be able to accommodate them due to conflicts with functional requirements of the system.
- There is a big gap between solutions and the requirements of the entire system.
Security Requirements Engineering

- Introduce security requirements analysis in the early phases of the software development process
- This allows us to
  - elicit security requirements from the organizational environment
  - analyze security requirements within the organizational environment in which the software will operate
  - motivate the use of specific security mechanisms
Object- vs Meta-Level Approaches

- **Object-level approach**
  - Use an off-the-shelf framework (e.g., UML, Kaos, i*/Tropos) as it is for modeling security requirements
    - Pro: modeling well established, reasoning feature ready
    - Con: modeling often cumbersome, some time impossible

- **Meta-level approach**
  - Take a RE framework and enhance it with novel constructs specific to security
    - Pro: modeling more effective and compact
    - Con: language constructs must gain “market acceptance”, semantics and reasoning to be update
Security vs Software Engineering

- **Software Engineer:**
  - design a system so that legitimate users can do what they want to do

- **Security Engineer:**
  - design a system so that illegitimate users cannot do what they should not do

- **Contentious Consequence**
  - We cannot use traditional Requirements or Software Engineering methodologies for Security, they have different overall goals!
Some Discarded Ideas…

- Discarded idea 1
  - Add primitives to Tropos/Kaos/UML/name-your-pet-RE-formalism to accommodate various security requirements
  - Confidentiality, authentication, access controls, etc., are security services and mechanisms NOT security requirements!

- Discarded idea 2
  - Model security requirements separately from functional requirements
  - Well, where’s the distinction then? Why bother?

- Discarded Idea 3
  - Model the goals of the attacker
  - They are not the goals of the security engineer!
Other Ideas

- **UML Proposals**
  - SecureUML, Model-Driven Architecture [Basin et al.]
  - UMLsec [Juriens]
  - Abuse Cases [McDermott & Fox]
  - Misuse Cases [Sindre & Opdhal]

- **Early Requirements Proposals**
  - Anti-requirements [van Lamsweerde et al., Crook et al.]
  - Problem-Frames, Abuse Frames [Hall et al., Lin et al]
  - Security Patterns [Giorgini & Mouratidis]
  - Privacy Modelling [Liu et al., Anton et al.]
Early Discussion

- UML Pros and Cons
  - Well-known even if meta-level extension not standardized
  - “Too Late” - model of system rather than organization

- Early requirements Pros and Cons
  - Capture organizational structure
  - “Too Functional” - Security is modelled explicitly and in parallel with the actual functional model
SecureUML

- D. Basin, J. Doser, and T. Lodderstedt, 2003
- They provide support for specifying access control policies
- The concepts of RBAC are represented as metamodel types
  - User, Role, Permission, Actions are types
  - UserAssignment, PermissionAssignment, RoleHierarchy are relations
  - AuthorizationConstraint is a predicate attached to a permission by the association ConstraintAssignment
    - Authorization constraints expressed in first-order logic
    - Used to establish the validity of the permission
SecureUML Metamodel

SecureUML Semantics

- An access control configuration is an assignment of users and permissions to role
- SecureUML makes access control decisions based on the access control configuration and on the validity of authorization constraints in a certain system state
  - Verify if a user is allowed to perform actions in the system state at a certain time with respect to an access control configuration
Limits of SecureUML

- NOT analyze security requirements within the organizational environment in which the software system will operate
- Require to know conflicting roles a priori
  - NOT detect conflicts from the requirements model of the system
Use Case Diagram

- Build a first sketch model of a system
- Characterize a way of using a system
- Offer a notation for describe the functionality of a system
  - Actors: an abstraction of an external agent that interact with the system
  - Use cases: specification of a type of interaction between a system and agents
  - Association lines: connect agents with the use cases in which they participate
Reservation System

Customer

Register customer

Add reservation

Delete reservation

Hotel Employee
Abuse Cases

- McDermott & Fox, 1999
- Negative use cases for modeling security requirements
- Specify 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 actors in the system
- Actors are the same that participate in use cases
Reservation System

- Customer
  - Steal credit card info
- Hotel Employee
  - Forget to introduce reservation
  - Introduce fake reservation
Limits of Abuse Cases

- Model security requirements separately from functional requirements
  - Abuse case diagrams show abuse only, not abuse together with normal use
  - They do not investigate relations between use and abuse
Misuse Cases

- Guttorm Sindre and Andreas Opdahl, 2000
- Extend use cases for modeling security requirements
- Specify behaviour that the system should avoid
- Specify how a misuser can damage the system
Concepts

- **Misuser**
  - hostile actor
  - a similar notation as an actor in use cases, except the misuser has a black "head" instead of white

- **Misuse case**
  - course of actions performed to do harm to a stakeholder or the system itself
  - behavior that is not wanted in the system
  - illustrated by black circles

- **Use cases**
  - functionalities of the system
  - countermeasures against misuse

- **Relations**
  - “includes” and “extends”
  - “prevents”: use case prevents the activation of a misuse case
  - “detects”: use case detects the activation of a misuse case
G. Sindre and A. Opdahl. 
Eliciting Security 
Requirements by Misuse 
Cases. In Proc. of TOOLS 
Pacific 2000.
Special mis-actors

Advantages

- Focus on security in the early phases of the software development process
- Increase the chance of discovering threats that otherwise would have been ignored
- Help to trace and organize the requirements specification
- Help to evaluate requirements
  - the real cost of implementing a use case includes the protection needed to mitigate all serious threats to it
- Easy to reuse in new development projects
Disadvantages

- Use/Misuse case are informal
  - No clear semantics
  - (Hence) NO formal analysis
- No knowledge on how to write *good quality* misuse cases
- The focus is ONLY on the system-to-be
- NOT suitable for all kinds of threats
- There is not always an identifiable misuser and the misuse case may not always consist of an identifiable sequence of actions
KAOS

- A research project
- Used to formalize goals into requirements
- Derive a description of a system's behavior
- Analyze system structure through acquiring and formalizing functional and non-functional requirements
- Supported by GRAIL tool
KAOS concepts

- **Agents**
  - active component of the system
  - play some role

- **Goals**
  - prescriptive statements of intent about the system
  - functional goals: service to be provided
  - non-functional goals: quality of service

- **Domain properties**
  - descriptive statements about the environment (e.g., physical laws, norms)
Goals

- Organized in terms of AND/OR hierarchies
  - Goal refinement used to construct a refinement-abstraction hierarchy
  - High level goals are strategic
    - Coarse grained with many agents
  - Low level goals are technical
    - Fine grained involving less agents
- Requirement: terminal goal for one agent
Obstacles

- Identify goal violation scenarios
- An obstacle to some goal is a condition whose satisfaction may prevent the goal from being achieved
- An obstacle $O$ is said to obstruct a goal $G$ in domain $Dom$ iff
  
  $\{O, Dom\} \models \neg G$ \hspace{1cm} Obstruction
  
  $Dom \not\models \neg O$ \hspace{1cm} Domain consistency
Obstacles Analysis

- Obstacles analysis consists in taking a pessimistic view of goals.
- Identify as many ways of breaking goals as possible in order to resolve each of such situations.
- Formal techniques for generation and AND/OR refinement of obstacles are available.
- Obstacles need to be resolved once they have been generated.
  - Resolution techniques: goal substitution, agent substitution, goal weakening, goal restoration, obstacle prevention and obstacle mitigation.
- Obstacle analysis is a recursive process.
  - it may produce new goals for which new obstacles may be generated and resolved.
Security Goals

- Considered a meta-class
- High level of abstraction
  - Confidentiality
  - Integrity
  - Availability
  - Privacy
  - Authenticity
  - Non-repudiation
- Each security goal has to be instantiated into application-specific security goal
Confidentiality

Goal Confidentiality

Goal Avoid [SensitiveInfoKnownByUnauthorizedAgent]

FormalSpec ∀ ag: Agent, ob: Object

¬ Authorized(ag,ob.Info) ⇒ ¬ Knows(ob.Info)

- If an agent ag is not authorized to access info about an object ob, then he does not knows any information info about the object ob

Goal Avoid [PaymentMediumKnownBy3rdParty]

FormalSpec ∀ p: Agent, acc: Account

¬ (Owns(p,acc) ∨ Manages(p,acc))

⇒ ¬ (Knows(acc.Acc#) ∧ Knows(acc.PIN))

- If agent p is not the owner of account acc and he should not manage it, he does not know number and PIN of the account
Obstacles Analysis

- Taking the negation of the goal
  \[ \forall p: \text{Agent}, \ acc: \text{Account} \]
  \[ (NG) \ \neg (\text{Owns}(p,acc) \lor \text{Manages}(p,acc)) \]
  \[ \land (\text{Knows}(acc.\text{Acc#}) \land \text{Knows}(acc.\text{PIN})) \]

- Suppose that the domain theory contains the following properties
  
  \( D1 \) \[ \forall p: \text{Agent}, \ acc: \text{Account} \ 	ext{Owns}(p,acc) \land \text{Knows}(p.\text{name}) \Rightarrow \text{Knows}(acc.\text{Acc#}) \]
  \( D2 \) \[ \forall acc: \text{Account} \ 	ext{Knows}(acc.\text{Acc#}) \Rightarrow \text{Knows}(acc.\text{PIN}) \]

- We can formally derived the following potential obstacle
  
  \( O \) \[ \forall p: \text{Agent}, \ acc: \text{Account} \]
  \[ \neg (\text{Owns}(p,acc) \lor \text{Manages}(p,acc)) \land \text{Knows}(p.\text{name}) \]
Anti-goals

- Obstacles sufficient for modeling and resolving non-intentional obstacles (accidental obstacles)
- Too limited for modeling and resolving intentional obstacles (malicious obstacles)
- Active attackers be modeled as well together with their own goals, capabilities, and the vulnerabilities they can monitor or control (anti-models)
- Anti-goals are the intentional obstacles to security goals
Building Anti-models

- Root anti-goal are obtained by negation of security goals
- For each anti-goal, potential attacker are identified (WHO)
- For each anti-goal and corresponding attacker, the higher level anti-goals are identified (WHY)
- For each anti-goal and corresponding attacker, the lower level anti-goals are identified (HOW)
- AND/OR refinement process for anti-goals
  - realizable by the attacker (anti-requirements)
  - realizable by the attacker (vulnerabilities)
- Anti-models are derived from anti-goals formulations
- Anti-requirements are defined in terms of the capabilities of the corresponding attacker
Running Antigoals

Goal *Avoid*[PaymentMediumKnownBy3rdParty]

*FormalSpec* \( \forall p: \text{Agent}, \ acc: \text{Account} \)
\[\neg (\text{Owns}(p,\text{acc}) \lor \text{Manages}(p,\text{acc})) \]
\[\Rightarrow \neg (\text{Knows}(\text{acc.Acc#}) \land \text{Knows}(\text{acc.PIN})) \]

AntiGoal *Achieve*[PaymentMediumKnownBy3rdParty]

*FormalSpec* \( \exists ag: \text{Agent}, \ ob: \text{Object} \)
\[\neg (\text{Owns}(p,\text{acc}) \lor \text{Manages}(p,\text{acc})) \]
\[\land \text{Knows}(\text{acc.Acc#}) \land \text{Knows}(\text{acc.PIN}) \]
Refining antigoals

By asking “what are sufficient conditions for someone unauthorized to know the number and PIN of an account simultaneously?”

**AntiGoal** *Achieve*[PaymentMediumKnownBy3rdPartyFromPinSearching]

**FormalSpec** \( \exists \) ag: Agent, ob: Object
\[ \neg (\text{Owns}(p,\text{acc}) \lor \text{Manages}(p,\text{acc})) \land \text{Knows}(\text{acc.}\text{Acc#}) \]
\[ \land (\exists \ x: \text{PIN}) [\text{Find}(p,x) \land \text{Match}(x,\text{acc.}\text{Acc#})] \]

**AntiGoal** *Achieve*[PaymentMediumKnownBy3rdPartyFromAccountNumberSearching]

**FormalSpec** \( \exists \) ag: Agent, ob: Object
\[ \neg (\text{Owns}(p,\text{acc}) \lor \text{Manages}(p,\text{acc})) \land \text{Knows}(\text{acc.}\text{PIN}) \]
\[ \land (\exists \ y: \text{Acc#}) [\text{Find}(p,y) \land \text{Match}(\text{acc.}\text{PIN},y)] \]
Limits of Antigoals

- Modeling attackers is difficult
- We have to consider all the possible obstacles even the ones unknown
  - Many protocols for security are been proved to be incorrect after some years they are designed
- Many system vulnerabilities depend on the particular implementation
- Software vulnerabilities are not completely known
Tropos and Security Requirements

- i*/Tropos has not been designed with security in mind
- Lack of the ability to capture at the same time functional and security features of an organization
- The process of integrating security and functional requirements throughout the whole range of the development stages is quite ad hoc
- The concept of softgoal that Tropos uses to capture security requirements fails to adequately capture some constraints that security requirements often represent
  - REMARK: softgoals are goals that have no clear-cut definition and/or criteria for deciding whether they are satisfied or not
- The methodology fails to provide concepts and processes to model trust relationships
Tropos Dependency

- A dependency between two actors means that the dependee will take responsibility for fulfilling the functional goal of a depender.
- Major assumption is that if you provide a service you have also the authority to decide who can use it, but...
- No way to specify or check whether the dependee is actually authorized to do so.
- It can happen that an actor depends on another for a service, but the dependee is neither the owner of the service nor authorized to provide the service.
Claim

- **Ownership and permission** are at the very foundation of all security concerns
  - no ownership, no security to worry about.
  - If people didn't own human rights, privacy rights, physical property, security would be a meaningless word
- Permission as a complementary notion to obligation is well-accepted in Deontic Logics
- So, we introduce it in our modeling framework
Moving towards a new proposal

- **Idea 1**
  - Security Requirements are social requirements
  - We need to start from a RE methodology modelling organizations
  - We need to capture the key social requirements for security

- **Idea 2**
  - We must model at the same time Functional Requirements and Security Requirements
  - So we can see the interplay of both and check one does not get in the way of the other

- **Occam's Razor**
  - Add few primitive constructs
  - Other security requirements as patterns, services, mechanisms
Secure Tropos

- Use the concepts offered by Tropos for actor, goal, task, and resource
- Make explicit who is the requester of the service, who is the legitimate owner of a service and who is able to provide a service
- Refinement of Tropos dependency
  - Trust relationship on Actor/service/Actor
  - Permission ≠ Execution
Requiring, Ownership, and Provisioning

- **Requiring**
  - Identify the objectives of actors

- **Ownership**
  - Identify actors who are the legitimate owners of goals, plans, or resources
  - The owner has full authority concerning the achievement of his goal, execution of his plan, or use of his resource
  - He can also delegate this authority to other actors

- **Provisioning**
  - Identify actors who have the capability to achieve goals, execute plans, or deliver resources
Delegation

- Delegation of permission
  - Used to model formal passage of authority
  - The delegatee thinks “I have the permission to fulfill the service (but I do not need to)”

- Delegation of execution
  - Used to model formal passage of responsibility
  - The delegatee thinks “Now, I have to get the service fulfilled”
Trust

- Trust is a relation between two actors representing the expectation of one actor (the trustor) about the capabilities and behavior of the other (the trustee)

- Trust of permission
  - the trustor believes that the trustee will not misuse the goal, task, or resource

- Trust of execution
  - the trustor believes that the trustee will achieve the goal, execute the task, or deliver the resource

- Trust is the mental counterpart of delegation
  - Delegation is an action due to a decision, whereas trust is a mental state driving such decision
Comparing Tropos and S-Tropos

- **Tropos Model**
  - Actor Properties
    - objectives
  - Actor Relationships
    - dependency

- **Secure Tropos Model**
  - Actor Properties
    - objectives
    - entitlements
    - capabilities
  - Actor Relationships
    - trust of execution
    - delegation of execution
    - trust of permission
    - delegation of permission
Comparing Tropos and S-Tropos

- **Dependency = Delegation of Execution + Trust of Execution**
  - if designer says A depends on B for G then A has actually delegated fulfillment of G to B and trusts that B will do it
  - if one depends on X to fulfill G, X is by default authorized to do G

- **Wanting = Owning**
  - if designer says that A wants G, of course A is authorized to fulfill G

- **Implicit Provisioning**
  - When designer stops dependency chain on goal G at agent B, it means that B will take care of it

- **Trust vs Delegation**
  - Permit to model scenario where actors must delegate permission or execution to other actors that do not trust
Secure Tropos -- Process

- Actor Diagram
  - Goals that an actors wants, provides, or owns
- Dependency model - Functional Requirements Model
  - Delegation of Execution
- Trust Model
  - Trust of Execution and Permission Relations
- Trust Management Implementation
  - Delegation of Permission
- Refinements by
  - Goal Decomposition within an Actor Diagram
  - Goal (Execution or Permission) Delegation to agents in a Delegation Diagram
  - Modification of Trust Relationship
Underlying Formal Model

- Formal Model
  - Answer Set Programming (aka Datalog¬)
  - Deduction, Satisfiability, Abduction
- Models (Secure Tropos Diagrams)
  - Extensional properties of classes (and instances)
- Axioms
  - Intensional properties and rules
- Properties
  - Specify conditions which must not be true in the model
  - Formulae that may be in true or may not be true
## Axioms

<table>
<thead>
<tr>
<th>Functional Requirement Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ax1:</strong> aims((B, S1)) :- depends((A, B, S1, S2))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Trust4Requirement Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ax2:</strong> trust((A, B, S1, S2, N - 1)) :- trust((A, B, S1, S2, N)) (\land N &gt; 2)</td>
</tr>
<tr>
<td><strong>Ax3:</strong> trust((A, C, S1, S2, P)) :- trust((A, B, S1, S2, N)) (\land) trust((B, C, S1, S2, M)) (\land P = \min{N - 1, M}) (\land N &gt; 2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Trust Management Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ax4:</strong> has((A, S)) :- owns((A, S))</td>
</tr>
<tr>
<td><strong>Ax5:</strong> has((B, S1)) :- delGrant((ID, A, B, S1, S2, N))</td>
</tr>
<tr>
<td><strong>Ax6:</strong> has((B, S1)) :- permission((ID, A, B, S1, S2))</td>
</tr>
<tr>
<td><strong>Ax7:</strong> fulfills((A, S)) :- has((A, S)) (\land) offers((A, S))</td>
</tr>
<tr>
<td><strong>Ax8:</strong> fulfills((A, S1)) :- depends((A, B, S1, S2)) (\land) fulfills((B, S1))</td>
</tr>
<tr>
<td><strong>Ax9:</strong> fulfills((A, S)) :- (\forall S' \subseteq S), fulfills((A, S'))</td>
</tr>
</tbody>
</table>
### Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pro1:</strong></td>
<td>$\text{aims}(A, S) \Rightarrow \text{fulfills}(A, S)$</td>
</tr>
<tr>
<td><strong>Pro2:</strong></td>
<td>$\text{has}(B, S_1) \land \text{owns}(A, S_1) \land A \neq B \Rightarrow \exists N \text{ trust}(A, B, S_1, S_2, N)$</td>
</tr>
<tr>
<td><strong>Pro3:</strong></td>
<td>$\text{fulfills}(B, S_1) \land \text{owns}(A, S_1) \land A \neq B \Rightarrow \exists N \text{ trust}(A, B, S_1, S_2, N)$</td>
</tr>
<tr>
<td><strong>Pro4:</strong></td>
<td>$\text{has}(B, S_1) \land \text{owns}(A, S_1) \land A \neq B \Rightarrow \begin{cases} \text{delGChain}(A, B, S_1, S_2) \lor \ \text{permissionChain}(A, B, S_1, S_2) \end{cases}$</td>
</tr>
<tr>
<td><strong>Pro5:</strong></td>
<td>$\text{fulfills}(B, S_1) \land \text{owns}(A, S_1) \land A \neq B \Rightarrow \text{permissionChain}(A, B, S_1, S_2)$</td>
</tr>
<tr>
<td><strong>Pro6:</strong></td>
<td>$\text{fulfills}(A, S) \Rightarrow \text{has}(A, S)$</td>
</tr>
<tr>
<td><strong>Pro7:</strong></td>
<td>$\text{permission}(ID, A, B, S_1, S_2) \Rightarrow \text{has}(A, S_1)$</td>
</tr>
<tr>
<td><strong>Pro8:</strong></td>
<td>$\text{delGrant}(ID, A, B, S_1, S_2, N) \Rightarrow \text{has}(A, S_1)$</td>
</tr>
<tr>
<td><strong>Pro9:</strong></td>
<td>$\text{permission}(ID, A, B, S_1, S_2) \Rightarrow \exists N \text{ trust}(A, B, S_1, S_2, N)$</td>
</tr>
<tr>
<td><strong>Pro10:</strong></td>
<td>$\text{delGrant}(ID, A, B, S_1, S_2, N) \Rightarrow \exists M \geq N \text{ trust}(A, B, S_1, S_2, M)$</td>
</tr>
<tr>
<td><strong>Pro11:</strong></td>
<td>$\text{permissionChain}(A, B, S_1, S_2) \Rightarrow \exists N \text{ trust}(A, B, S_1, S_2, N)$</td>
</tr>
<tr>
<td><strong>Pro12:</strong></td>
<td>$\text{delGChain}(A, B, S_1, S_2) \Rightarrow \exists N \text{ trust}(A, B, S_1, S_2, N)$</td>
</tr>
<tr>
<td><strong>Pro13:</strong></td>
<td>$\text{delGChain}(A, B, S_1, S_2) \Rightarrow \begin{cases} \exists M \ \exists A_1 \ldots A_M \ \exists N_1 \ldots N_{M-1} \ \forall i \in [1 \ldots M - 1] \ \text{delGrant}(ID_i, A_i, A_{i+1}, S_1, S_2, N_i) \land \ A_1 = A \land A_M = B \land N_i &gt; N_{i+1} \end{cases}$</td>
</tr>
</tbody>
</table>
Requirements Analysis Process
Computer-Aided SRE

- Draw the graphical (Secure) Tropos models
- Diagrams (automatically) mapped into a Formal Model
  - Datalog specifications
  - Formal Tropos specification
- Check the properties on the Formal Model
  - Integration within different Datalog solvers
    - DLV System, ASSAT, C-Models, S-Models
Secure Tropos Tool
Social vs Individual

- Tropos involves two different levels of analysis
  - Social level: the structure of organizations are defined associating to every role (or position) objectives and responsibilities
  - Individual level: agents are not only defined with their objectives and responsibilities, but also they are associated to roles (or positions) they can play
- In Tropos there is no explicit separation between the two levels, and it is very difficult to maintain the consistency
Social vs Individual (2)

- It is possible that requirements are given only at individual level or at social level
- Social => Individual
  - The agents playing a social role “should” inherit properties and social relations of that role
  - If Alice plays R1 and R1 trusts R2 and Bob plays R2 then Alice trusts Bob…
- Useful feature to “complete” models in Computer aided RE
  - Social relationships are always drawn in RE
    - After all they are among the system specifications
  - Designers must only draw social relationships and the reasoning system does the rest
Distrust

- Need for negative authorizations to help designers in shaping the perimeter of positive trust
  - Distrust is a relation between two actors representing the expectation of one actor about the *incapabilities* and *misbehaviour* of the other
- Used to identify illegitimate actors
- Distrust as a primitive
  - Model Trust and Distrust as independent primitive
  - Distrust as absence of trust = Close World Assumption
- Trust Conflicts
  - The presence of positive and negative authorization at the same time could generate conflicts on trust relationships
  - Computer Aided RE automatically detects such conflicts
Sample Conflicts

Distrust at social level (e.g., procedures imposing restriction on roles)
Trust at individual level

Trust at social level
Distrust at individual level
Verification Process

- Design models at both social level and individual level, independently
- Verify correctness and consistency of social level
- Map relations at social level into models at individual level
- Solve conflicts if needed
- Verify correctness and consistency of models at individual level
References


Tropos project: http://www.troposproject.org