International Journal of Software Engineering and Knowledge Engineering
© World Scientific Publishing Company

MODELLING SECURE SYSTEMS USING AN AGENT-ORIENTED APPROACH AND SECURITY PATTERNS

HARALAMBOS MOURATIDIS
School of Computing and Technology, University of East London,
Barking Campus, Longbridge Road, RM8 2AS, England
h.mouratidis@uel.ac.uk

MICHAEL WEISS
School of Computer Science, Carleton University,
1125 Colonel By Dr, Ottawa, Ontario K1Y 0Z3, Canada
weiss@scs.carleton.ca

PAOLO GIORGINI
Department of Information and Communication Technology, University of Trento,
Via Sommarive 14, I-38050, Povo, Trento, Italy
paolo.giorgini@dit.unit.it

In this paper we describe an approach for modelling security issues in information systems. It is based on an agent-oriented approach, and extends it with the use of security patterns. Agent-oriented software engineering provides advantages when modeling security issues, since agents are often a natural way of conceptualizing an information system, in particular at the requirements stage, when the viewpoints of multiple stakeholders need to be considered. Our approach uses the Tropos methodology for modelling a system as a set of agents and their social dependencies, with specific extensions for representing security constraints. As an extension to the existing methodology we propose the use of security patterns. These patterns capture proven solutions to common security issues, and support the systematic and structured mapping of these constraints to an architectural model of the system, in particular for non-security specialists.

Keywords: Agent-oriented software engineering, security, patterns, Tropos

1. Introduction

Modern information systems support critical worldwide infrastructures such as transportation, finance and communication. Consequently, critical and sensitive information is stored on such systems. As a result, it is widely accepted that security is of particular importance to these information systems. Moreover, it has been argued that in order to help towards the development of secure information systems, security concerns must inform all phases of the development process of an information system.[12] However, dealing with security concerns is not simple. This is mainly due to the following reasons: (1) non-security experts are involved in the development of systems that require knowledge of security; (2) many different concepts are used between security specialists and software engineers; (3) there is an ad hoc approach towards security; (4) both systems and security engineering are quite complex; and (5) it is difficult to fully test the proposed solutions at the design level.

As a result, researchers have initiated work to address these problems (see Section 7 for an overview). However, all of these approaches consider security as a one-dimensional problem, that is, they only provide solutions to isolated problems. We believe that the integration of agent-oriented software engineering and security patterns represents an effective solution to this concern. That is mainly due to the appropriateness of agent-oriented approaches for dealing with security issues in information systems, and the appropriateness of patterns for transferring security-related knowledge to developers that are not security experts.

Security requirements are mainly obtained by analysing an organization's attitude towards security and after studying its security policy. An agent-oriented perspective allows us to model the objectives of multiple stakeholders and their interactions, and to analyze how security requirements propagate to the rest of the system. In addition, an agent-oriented view is perhaps the most natural way of characterising security issues in software systems. Characteristics, such as autonomy, intentionality and sociality, provided by the use of agent-orientation allow developers to first model the security requirements at a high level, and then incrementally transform these high-level requirements to security mechanisms.

In our previous work we described an extension to the agent-oriented software engineering methodology Tropos for modelling security issues in information systems.[21,22] In this paper we propose the use of security patterns as a complement to this methodology. Security patterns capture design experience and proven solutions to security-related problems in such a way that can be applied by non-security experts.[13] Security patterns also prevent ad hoc solutions by helping apply proven solutions in a systematic and structured way. In addition, they introduce abstraction layers which help closing the gap between security experts and developers.

The rest of the paper is structured as follows. Section 2 describes the agent-oriented development methodology Tropos enhanced with security concepts to enable it to model security issues throughout the development phases of an information system. In Section 3 we introduce a case study, the electronic Single Assessment Process (eSAP) system — an integrated health and social care information system for the assessment of the needs of older people for which security is an important concern — that demonstrates the use of these security extensions. In Section 4 we present a set of security patterns, and in Section 5 we demonstrate, using the eSAP system as an example, how these patterns can be applied within the architectural design phase of the Tropos methodology. Section 6 provides a formal model of the completeness of the pattern language. Section 7 presents a discussion of related work, and Section 8 concludes the paper and outlines directions for future work.

2. Agent-Orientation and the Tropos Methodology

Agent-Oriented Software Engineering (AOSE) is emerging as a powerful, new paradigm for the development of information systems. Its major modeling construct is the *agent*, which demonstrates properties such as autonomy, intentionality, sociality, identity and boundaries, strategic reflectivity, and rational self-interest.[29] As a result, agent-orientation provides a higher level of abstraction than other software development paradigms, such as object-orientation. However, an important point of agent-oriented software engineering is that its use for the analysis and design of a system does not necessarily impose the use of agents in the implementation.

Tropos is a development methodology tailored to the description of the organisational environment of a system and the system itself.[6] Tropos emphasizes the early requirements analysis that precedes requirements specification, addressing the need to understand how and why the intended system would meet the organisational goals. This allows for a more refined analysis of system dependencies, leading to a better treatment not only of the system's functional requirements but also of its non-functional requirements, such as security, reliability, and performance.

Tropos adopts the *i** modelling framework, which uses the concepts of actors, goals, tasks, resources and social dependencies for defining the obligations of actors (dependees) to other actors (dependers).[28] *Actors* have strategic goals and intentions within the system or the organisation and represent (social) agents (organisational, human or software), roles or positions (a set of roles). A *goal* represents the strategic interests of an actor. We differentiate between hard goals (or simply goals) and soft goals. Soft goals represent non-functional requirements and have no clear definition or criteria for deciding on whether they are satisfied or not. An example of a soft goal is "the system should be scalable". A *task* represents a way of doing something. Thus, for example a task can be executed in order to satisfy a goal. A *resource* represents a physical or an informational entity, while a *social dependency* between two actors indicates that one actor depends on another to accomplish a goal, execute a task, or deliver a resource. Figure 1 shows the notation for these concepts.

Although Tropos was not conceived with security in mind, a set of security concepts, such as security constraint, secure entities and secure dependencies have been proposed by Mouratidis that allow Tropos to model security aspects throughout the full development process.[21,22] A *security constraint* is defined as a constraint that is related to the security of the system, while *secure entities* represent any secure goals/tasks/resources of the system. *Secure goals* are introduced to the system to help in the achievement of a security constraint. A secure goal does not define specifically how the security constraint can be achieved, since (as in the definition of a goal) alternatives can be considered. However, this can be modeled as a *secure task*, which secure task represents a particular way for satisfying a secure goal.

A *secure dependency* introduces security constraints, proposed either by the depender or the dependee in order to successfully satisfy the dependency. Both the depender and the dependee must agree on these constraints for the dependency

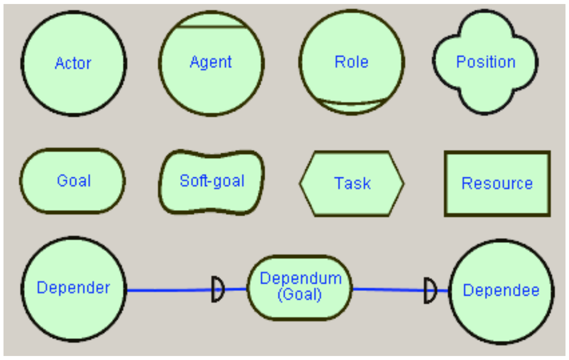

Fig. 1. Notation used by the Tropos modelling framework. It uses the concepts of actors, goals, tasks, resources and social dependencies for defining the obligations between actors.

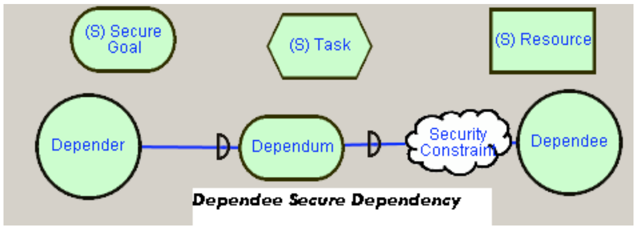

Fig. 2. Extensions to the Tropos notation for modelling security concepts.

to be valid. The *depender* expects the *dependee* to satisfy the security constraints, while the dependee will make an effort to deliver the *dependum* by satisfying the constraints. The security concepts added to Tropos are shown in Figure 2.

Tropos covers five main software development phases:[6]

- early requirements analysis, concerned with the understanding of a problem by studying an existing organisational setting;
- late requirements analysis, where the system is described in its operating environment, along with relevant functions and security requirements;
- architectural design, where the global system architecture is defined in terms of subsystems, interconnected through data and control flows;
- detailed design, where each architectural component is defined in terms of inputs, outputs, control, and security aspects; and
- implementation, during which the system components are implemented according to the previous phases (not necessarily agent-based).

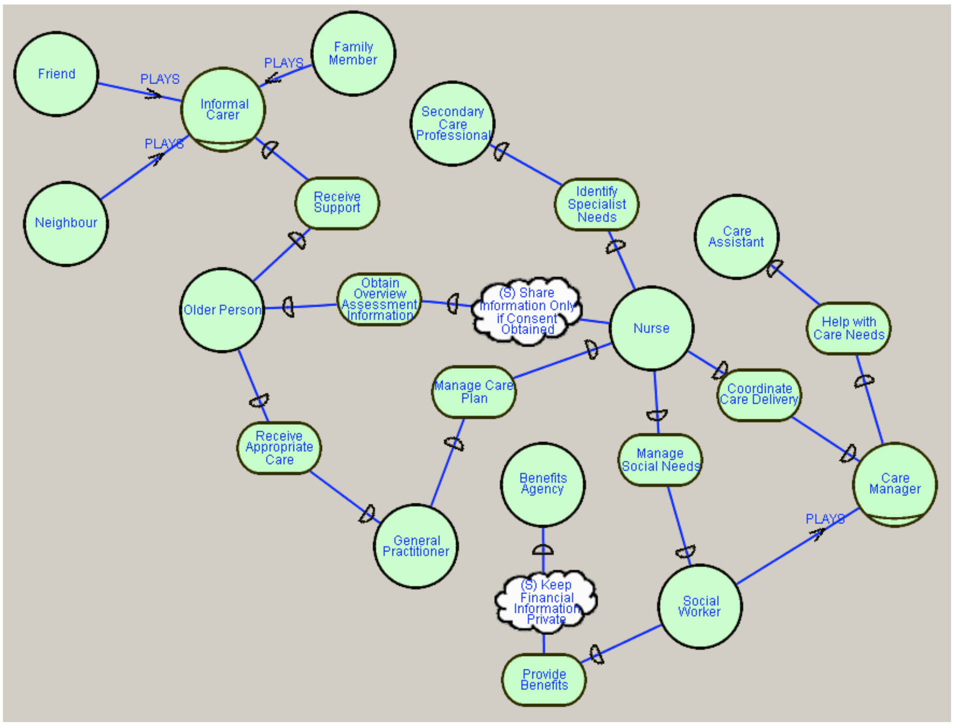

Fig. 3. Actor diagram for the eSAP system.

3. Using the Security Extensions of the Tropos Methodology

To demonstrate how the above security concepts and procedures can be used in the development of secure information systems, we consider the electronic Single Assessment Process (eSAP) system, an integrated health and social care information system for the effective care of older people.[23] Security is an important concern for eSAP, since security breaches of such a system might result in personal and health information to be revealed, which could lead to serious consequences.

3.1. *Early Requirements Analysis*

During the early requirements analysis phase, the goals, dependencies and the security constraints between the stakeholders (actors) are modeled with the aid of an actor diagram.[6] Such a diagram involves actors and the dependencies between the actors. Some of the actors involved in the eSAP system along with their dependencies and security constraints are shown in Figure 3.

However, an actor diagram does not provide an analysis of the individual actor's goals and the security constraints imposed on them. This information can be modelled with the aid of rationale diagrams.[6] In a rationale diagram, an actor's goals

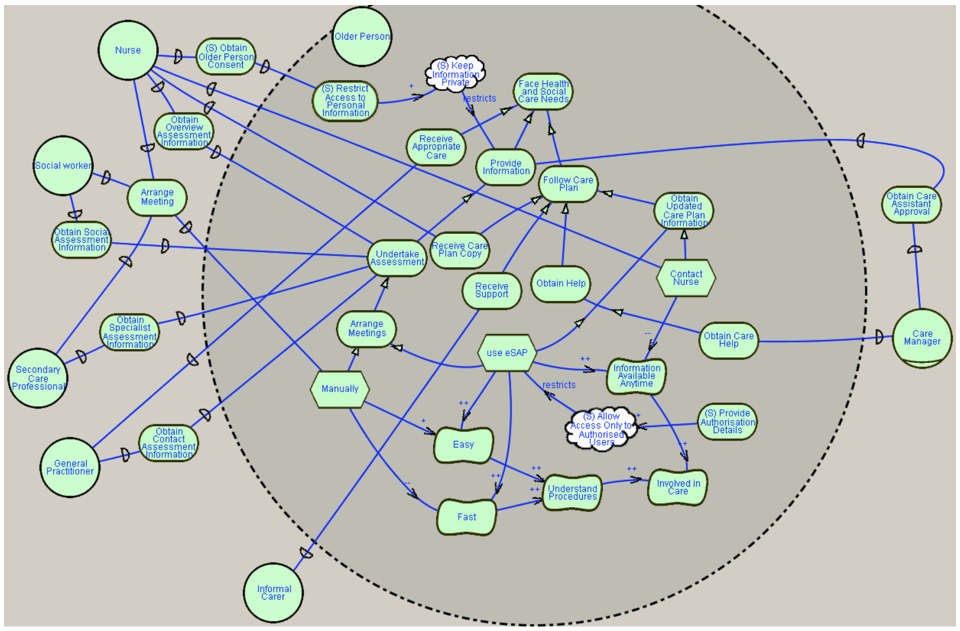
Nurse
(S) Obtain Older Person Consent
Older Person
(S) Keep Information Private
(S) Restrict Access to Personal Information
Face Health and Social Care Needs
restricts
Obtain Overview Assessment Information
Receive Appropriate Care
Provide Information
Social worker
Arrange Meeting
Follow Care Plan
Obtain Updated Care Plan Information
Obtain Care Assistant Approval
Obtain Social Assessment Information
Undertake Assessment
Receive Care Plan Copy
Contact Nurse
Receive Support
Obtain Help
Obtain Specialist Assessment Information
Arrange Meetings
use eSAP
Obtain Care Help
Care Manager
Secondary Care Professional
Information Available Anytime
Manually
restricts
(S) Allow Access Only to Authorised Users
(S) Provide Authorisation Details
Easy
General Practitioner
Obtain Contact Assessment Information
Understand Procedures
Involved in Care
Fast
Informal Carer

Fig. 4. Rationale diagram for the Older Person actor.

and security constraints are explicitly analysed. A dashed circle encloses the actor. Figure 4 illustrates the rationale diagram of the Older Person actor.

3.2. *Late Requirements Analysis*

Once the actors have been analysed and their goals, dependencies and security constraints identified, we can proceed to the late requirements analysis phase. In this phase, the functional, security, and other non-functional requirements for the system to be are elaborated. The system to be is introduced as one or multiple actors that have a number of dependencies with the other actors of the organization (as defined during the early requirements phase). For example, the eSAP system contributes to the goals of the stakeholders as shown in Figure 5.

The introduced system (the eSAP system in our case study) is further analysed using the same concepts used for the analysis of the other actors. Figure 6 shows a rationale diagram of the eSAP system. To satisfy the security objectives of the system, different security constraints are imposed. In our example, the security constraints have been derived from the security policy for medical information systems identified by Anderson.[2] Then, the security constraints are further analysed according to security constraint analysis.[21,22]

For instance, by analysing the Keep System Data Private security constraint in Figure 6, we derive that tasks such as Check Access Control, Check Authentication,

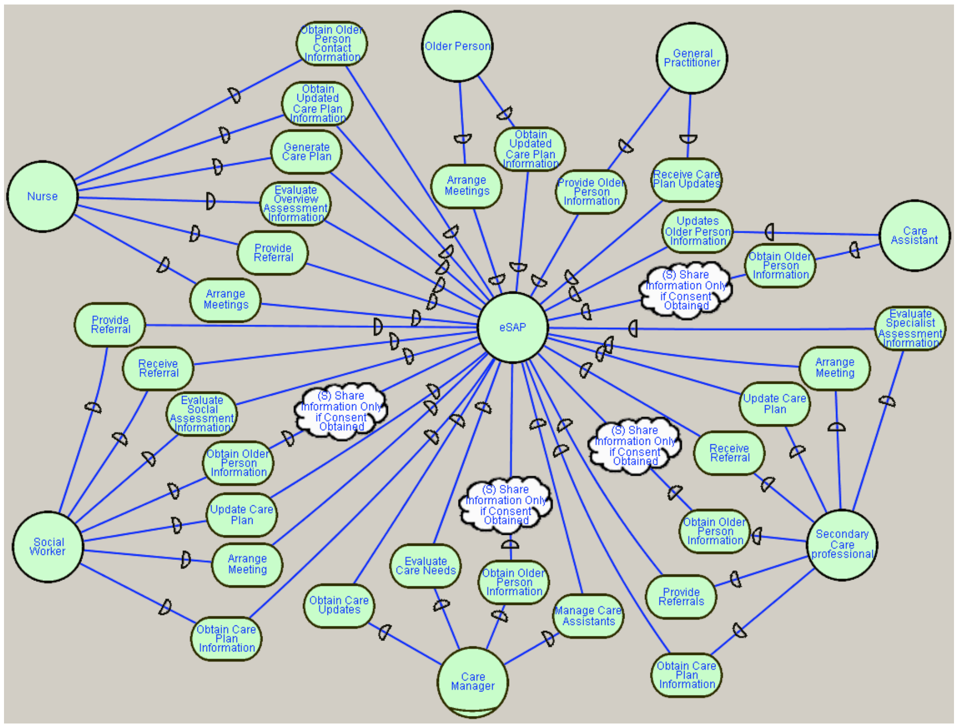
Obtain Older Person Contact Information
Older Person
General Practitioner
Obtain Updated Care Plan Information
Generate Care Plan
Obtain Updated Care Plan Information
Arrange Meetings
Provide Older Person Information
Receive Care Plan Updates
Nurse
Evaluate Overview Assessment Information
Provide Referral
Updates Older Person Information
Care Assistant
Obtain Older Person Information
(S) Share Information Only if Consent Obtained
Arrange Meetings
eSAP
Provide Referral
Evaluate Specialist Assessment Information
Receive Referral
Arrange Meeting
Evaluate Social Assessment Information
(S) Share Information Only if Consent Obtained
Update Care Plan
Obtain Older Person Information
(S) Share Information Only if Consent Obtained
Receive Referral
Update Care Plan
(S) Share Information Only if Consent Obtained
Obtain Older Person Information
Social Worker
Arrange Meeting
Secondary Care professional
Evaluate Care Needs
Obtain Older Person Information
Manage Care Assistants
Provide Referrals
Obtain Care Updates
Obtain Care Plan Information
Care Manager
Obtain Care Plan Information

Fig. 5. Actor Diagram including the eSAP system as an actor.

and Check Information Flow must be achieved in order to fulfill this security constraint. Each of those tasks can be achieved by considering different alternatives. For example, Check Authentication can be achieved in three different ways: Check Password, Check Digital Signature, or Check Biometrics.

The introduced system (the eSAP system in our case study) is further analysed using the same concepts used for the analysis of the other actors. Figure 6 shows a rationale diagram of the eSAP system. To satisfy the security objectives of the system, different security constraints are imposed. In our example, the security constraints have been derived from the security policy for medical information systems identified by Anderson.[2] Then, the security constraints are further analysed according to security constraint analysis.[21,22]

4. Security Patterns for Multiagent Systems

Using the security-oriented extension of the Tropos methodology we have identified the security requirements of the system during early and late requirements analysis. The next phase of the methodology is the *architectural design* phase. During this phase, the requirements are transformed into a design. However, as discussed in

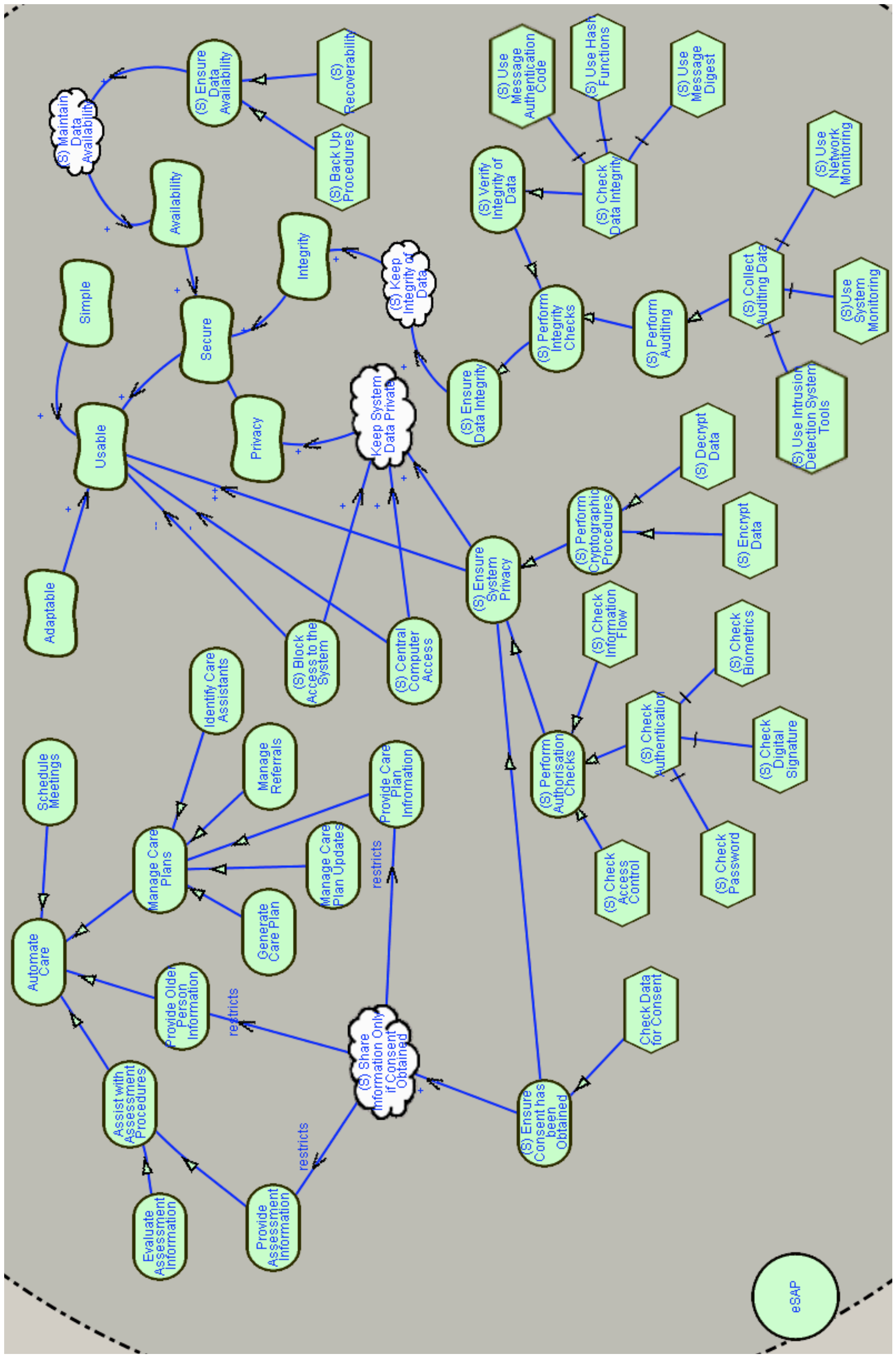
(S) Maintain Data Availability
(S) Ensure Data Availability
(S) Recoverability
(S) Back Up Procedures
Availability
(S) Use Message Authentication Code
(S) Use Hash Functions
(S) Use Message Digest
(S) Verify Integrity of Data
(S) Check Data Integrity
(S) Use Network Monitoring
Integrity
(S) Keep Integrity of Data
Simple
Secure
(S) Perform Integrity Checks
(S) Perform Auditing
(S) Collect Auditing Data
(S)Use System Monitoring
(S) Ensure Data Integrity
Keep System Data Private
Usable
Privacy
(S) Use Intrusion Detection System Tools
(S) Decrypt Data
(S) Perform Cryptographic Procedures
(S) Encrypt Data
Adaptable
(S) Ensure System Privacy
(S) Check Information Flow
(S) Check Biometrics
(S) Block Access to the System
(S) Central Computer Access
Identify Care Assistants
Schedule Meetings
Manage Referrals
Provide Care Plan Infromation
(S) Check Authentication
(S) Check Digital Signature
(S) Perform Authorisation Checks
Manage Care Plans
Manage Care Plan Updates
restricts
(S) Check Access Control
(S) Check Password
Automate Care
Generate Care Plan
Provide Older Person Information
restricts
Check Data for Consent
(S) Share Information Only if Consent Obtained
Assist with Assessment Procedures
(S) Ensure Consent has been Obtained
restricts
Evaluate Assessment Information
Provide Assessment Information
eSAP

Fig. 6. Rationale diagram for the eSAP system.

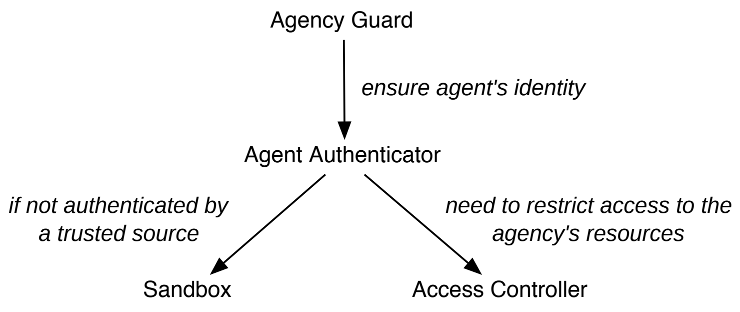
Agency Guard
ensure agent's identity
Agent Authenticator
if not authenticated by a trusted source
need to restrict access to the agency's resources
Sandbox
Access Controller

Fig. 7. Roadmap of the pattern language for secure agent systems

Section 1 for a developer without knowledge of security this could be a very difficult task, possibly resulting in the development of a non-secure system. For this reason we introduce *security patterns*. Patterns capture existing proven experience in software development and help to promote best design practices.[9]

Patterns are often organized in the form of pattern languages. A *pattern language* is a set of closely related patterns that guides the developer through the process of designing a system. Using a pattern language, a design starts as a “fuzzy cloud” that represents the system to be realized. As patterns are applied, parts of the system come into focus, each pattern suggesting new patterns to be applied that refine the design, until no more patterns can be applied.[5]

A pattern language for the development of secure agent-based systems should employ agent-oriented concepts, such as intentionality, autonomy, sociality and identity. The structure of a pattern should be described not only in terms of collaborations and the message exchange between the agents, but also in terms of their social dependencies and intentional attributes, such as goals and tasks. This allows for a complete understanding of the pattern’s social and intentional dimensions.

Our pattern language has four patterns: *Agency Guard*, *Agent Authenticator*, *Sandbox*, and *Access Controller*. Figure 7 provides a “roadmap” of the pattern language. The arrows in the roadmap show dependencies between patterns, and point from one pattern to the patterns that developers may want to consult once this pattern has been applied. The roadmap thus suggests to begin the architectural design with the *Agency Guard* pattern. The annotations on the arrows summarize the rationale for selecting a pattern in the context of another pattern.

We use the Alexandrian format for organizing each pattern.[3] In this format, the sections of a patterns are context, problem and forces, solution, and rationale, each section set off from the next by a set of stars. Brief descriptions of the problem and solution are put in boldface, followed by more detailed discussions. The rationale section is organized into benefits, liabilities, and related patterns.

4.1. *Agency Guard*

... a number of agencies exist in a network. Agents from different agencies must communicate with each other, or exchange information. This involves the movement of some agents from one agency to another, or requests from agents belonging to one agency for resources belonging to another agency.

***

A malicious agent that gains unauthorized access to the agency can disclose, alter or, generally, destroy data residing in the agency.

Many malicious agents will try to gain access to agencies that they are not allowed to access. Depending on the level of access the malicious agent gains, it might be able to completely shut down the agency, or exhaust the agency's computational resources, and thus deny services to authorised agents. The problem becomes the more severe the more backdoors there are to an agency, enabling potential malicious agents to attack the agency from many places. On the other hand, not all agents trying to gain access to the agency must be treated as malicious, but rather access should be granted based on the security policy of the agency.

Therefore:

Ensure that there is only a single point of access to the agency.

When a Requester Agent wishes to access resources of an Agency or move to this agency, its requests must be forwarded through an Agency Guard that is responsible for granting or denying access requests according to the security policy of the agency. The Agency Guard is the only point of access to the Agency, and cannot be bypassed, that is, all access requests must go through the Agency Guard. In traditional terms, the concept of an Agency Guard is referred to as a monitor.[1]

The structure of the pattern in terms of the actors involved and their social dependencies is shown in Figure 8. The remaining dependencies are as follows. The Agency depends on the Agency Guard to grant or deny access to the Agency. The Agency Guard will grant or deny access according to the security policy of the Agency, and depends on the Agency to obtain this security policy.

***

Benefits:

- Only the Agency Guard needs to be aware of the security policy, and it is the only entity that must be notified if the security policy changes.
- Being the single point of access, only the Agency Guard must be tested for correct enforcement of the agency's security policy.

Liabilities:

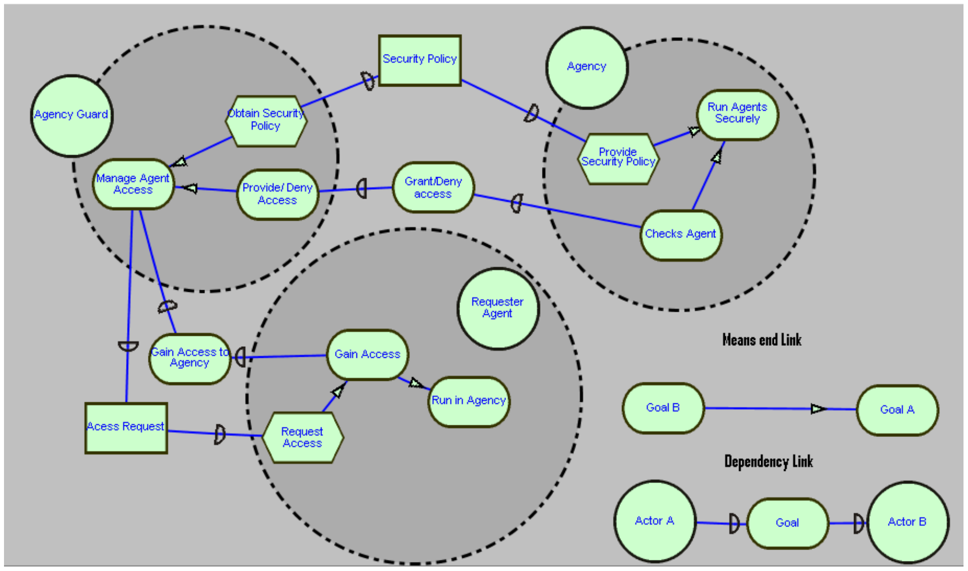

Fig. 8. Structure of the *Agency Guard* pattern.

- A single point of access to the agency can degrade the performance of the agency (that is, the response time for handling access requests).
- The Agency Guard is a single point of failure. If it fails, the security of the agency as a whole is at risk.

Related patterns:

- *Agent Authenticator* – ensures the identity of the Requester Agent.

4.2. *Agent Authenticator*

... you are using *Agency Guard* to protect access to an agency or its resources. To be allowed access, agents must be authenticated, that is, they must provide information about the identity of their owners.

***

Many malicious agents will try to masquerade their identity when requesting access to an agency.

If such an agent is granted access to the agency, it might try to breach the agency's security. In addition, even if the malicious agent fails to cause problems in the security of the agency, the agency under attack will no longer trust the agent impersonated by the malicious agent.

Therefore:

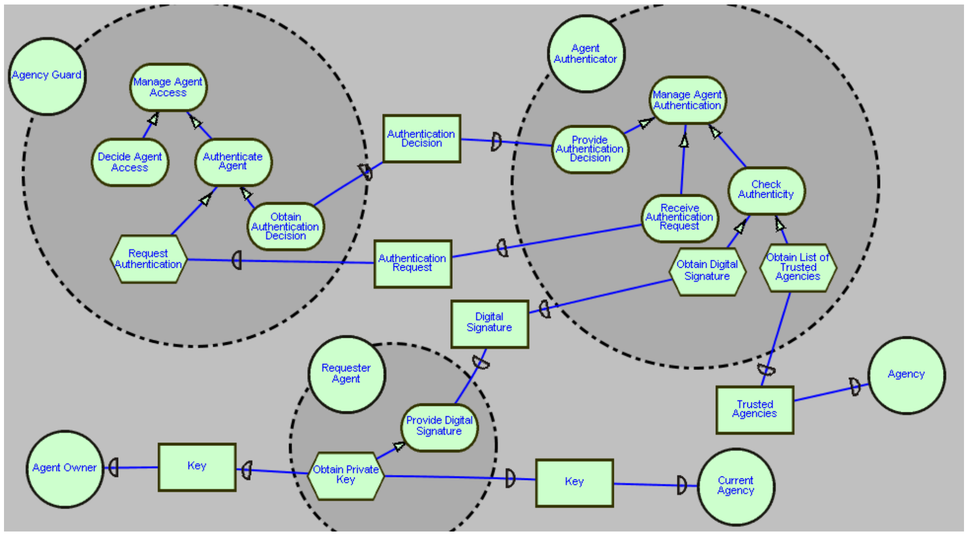

Fig. 9. Structure of the *Agent Authenticator* pattern.

Authenticate agents as they enter the agency.

Requester Agents have to be authenticated by the Agency. By authenticating the agent, the Agency Guard makes sure it comes from an owner that is trusted by the Agency. Each Requester Agent's owner and each Agency have a public/private key pair. The Agent Authenticator can authenticate the Requester Agent in two ways: the agent can be digitally signed with the owner's private key, or with the private key of the Agency in which the agent resides. In order for the second approach to work, mutual trust must be established between the sending and receiving agencies (each Agency can be set up so it has a list of trusted agencies). If the Agent Authenticator does not trust the Agency from which the agent originates, it can reject the agent, or accept it with minimal privileges and execute it in a *Sandbox*.

The structure of the pattern is shown in Figure 9. The Agency Guard depends on the Agent Authenticator to authenticate the agent, and, in turn, the Agent Authenticator has to receive the request from the Agency Guard. The Agent Authenticator has to send the notification to the Agency Guard once the agent is authenticated.

***

Benefits:

- Since authentication concerns are dealt with in a single location, it is not necessary to provide each agent with its own authentication mechanism.
- The use of an Agent Authenticator ensures that Requester Agents are authenticated, before they can request a resource from the agency.

- When implementing the system, only the Agent Authenticator must be checked for correct enforcement of the agency's security policies.

Liabilities:

- The Agent Authenticator is a single point of failure. If it fails, the security of the agency as a whole is at risk.

Related patterns:

- *Sandbox* – allows running an agent that could not be authenticated with minimal privileges.
- *Access Controller* – restricts access to the agency's resources.

4.3. *Sandbox*

... you are using *Agent Authenticator* to ensure the requester agent's identity, but the requester agent cannot be properly authenticated. This can be the case either when the agent could not be authenticated, or if it has been authenticated by an agency that the receiving agency does not trust.

***

An agency is most likely exposed to a large number of malicious agents that will try to gain unauthorized access to it.

Although the agency will try to prevent access to those agents, it is possible that some of them might be able to gain access to the agency's resources. Thus, it is necessary for the agency to operate in a manner that will minimize the damage which can be caused by unauthorized agents gaining access. In addition, some unauthorized agents might be allowed access by the agency in order to provide services the agency's agents cannot provide. Thus, the agency must be cautious to accept such unauthorized agents without putting its security at risk.

Therefore:

Execute the agent in an isolated environment that has full control over the agent's ingoing and outgoing messages.

Implementing this principle prevents any malicious code from doing something it is not authorised to do. The code is allowed to destroy anything within a restricted environment (a *sandbox*), but it cannot touch anything outside. The concept is similar to the Java security model, and the `chroot` environment in UNIX. The Sandbox observes all system calls made by the code, and compares them to the agency-defined policy. If any violations occur, the Agency can shut down the suspicious agent.

The structure of the pattern is shown in Figure 10. The Agency depends on the Sandbox to observe and control the Requester Agent's activities, and the Sandbox depends on the Agency to know the agency's policies for agents sent to the Sandbox.

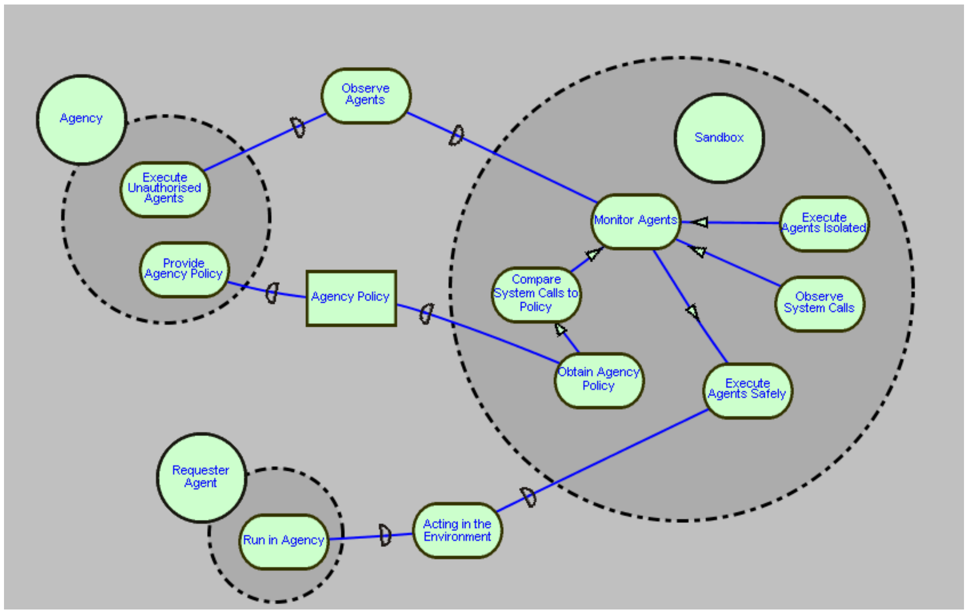

Fig. 10. Structure of the *Sandbox* pattern.

***

Benefits:

- Agents not authorised but valuable for the agency can be executed without compromising its security.
- The agency can identify possible attacks (by observing the actions of the agents in the sandbox).

Liabilities:

- Some computational resources of the agency might be diverted to non-useful actions, if non-useful agents are sandboxed.
- The use of a sandbox introduces an extra layer of complexity.

Related patterns:

- N/A

4.4. *Access Controller*

... you are using *Agent Authenticator* to ensure the requester agent's identity. Now you need to restrict access to the agency's resources. Many different agents can exist in an agency, which require access to the agency's resources in order to achieve their operational goals. However, they should can only access specific resources.

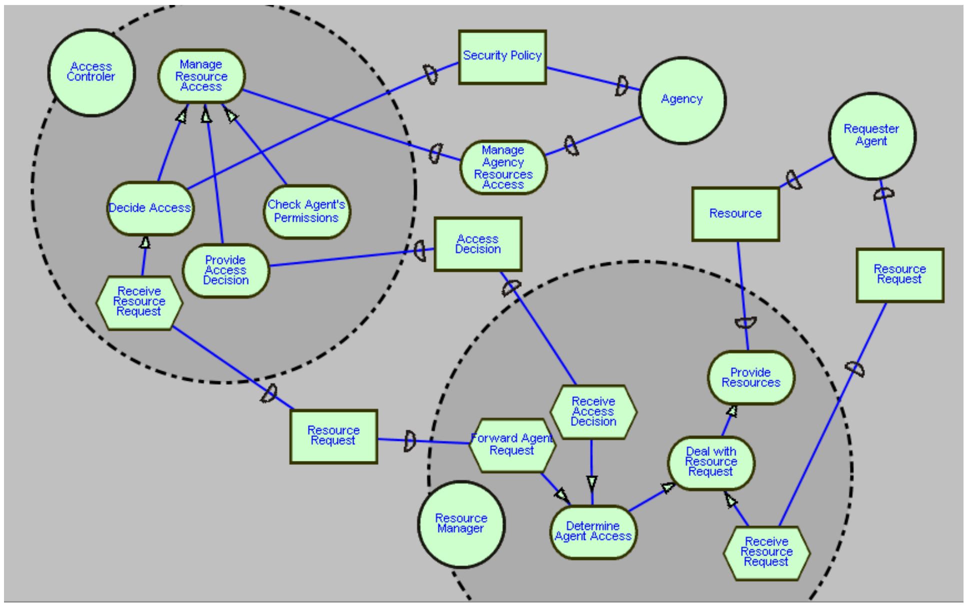

Fig. 11. Structure of the *Access Controller* pattern.

***

Agents belonging to an agency might try to access resources that they are not allowed to access.

Allowing this to happen might lead to serious problems such as the disclosure of private information, or the alteration of sensitive data. In addition, different security privileges will be applied to different agents in the agency. The agency should take into account its security policy and consider each access request individually.

Therefore:

Intercept all requests for the agency's resources.

The agency uses an Access Controller to restrict access to each of its resources. Thus, when a Requester Agent requests access to a resource, the request is dispatched to the Access Controller, which then checks the security policy, and determines whether the access request should be approved or rejected. Only if the request is approved, is the request forwarded to the corresponding Resource Manager.

The structure of the pattern is shown in Figure 11. The Requester Agent depends on the Resource Manager for the resource, and the Agency depends on the Access Controller for checking the request. The Access Controller, in turn, depends on the Agency for receiving the security policy and for forwarding the request, which it forwards to the Resource Manager in case the request is approved.

***

Benefits:

- The agency's resources are used only by agents allowed to access them.
- Different policies can be used for accessing different resources.

Liabilities:

- There is a single point of attack. If the Access Controller is compromised, the system's access control system fails.

Related patterns:

- N/A

4.5. *Qualitative Evaluation of the Pattern Language*

An important question that might be raised is *how well* the proposed language prevents developers from building systems that contain security holes. One answer to this question is to evaluate how well our language follows the guiding principles for secure information systems design developed by Viega and McGraw.[26]

In accordance with these principles, the different patterns of the language allow developers to break up the security of the system into different components (*principle of comparmentalization*) that are simple to develop and manage (*principle of keeping the system simple*). For example, the Agency Guard pattern indicates that there is only one point of access to the system, considerably reducing the design effort required if there was more than one way to access the system.

Moreover, the application of the patterns of the language to the development of a system ensures that in case of failure the system will fail safely (*principle of failing securely*), since if one component fails, security is still achieved with the rest of the patterns. For example, if the Authenticator fails, the Sandbox pattern ensures that any agent arriving at the system will be running in a restricted environment without any privileges that might endanger the security of the system.

To protect a system, the pattern language proposes different levels of security including authentication, access control, and sandboxing, and as a result it promotes the *principle of practicing defence in depth* by avoiding a monolithic solution that would provide security only at one level of defence. The Access Controller pattern ensures that agents are allowed access only to resources they need, and as a result it practices the *principle of least privilege*. In addition, the Authenticator pattern ensures that only authorised entities have access to specific information, thus use of the pattern satisfies the *principle of promoting privacy*.

Finally, the language adheres to the *principle of community resources* by employing patterns that were derived from well-tested agent security solutions.

Although these principles cannot guarantee 100% security (no approach can guarantee that), their authors estimate that they cover about 90% of all potential problems by providing help in three different ways: by *preventing* common errors during the development process; by *coping* with unknown attacks; and by *facilitating* the understanding of security patterns and providing security insight.

5. Applying the Security Patterns

In this section we describe how the security patterns can be employed during the architectural design stage of the Tropos methodology. The integration of the patterns at this stage of the methodology will help identify additional actors to fulfill the security goals of the system without putting its security at risk.

During architectural design the global system architecture is defined in terms of subsystems (actors) and their interconnection through data and control flows. From a security viewpoint, an important step of this stage is to identify the actors responsible for achieving the system's security goals. Security patterns can greatly help identifying those actors without putting the security of the system at risk.

5.1. *Guidelines for Applying the Pattern*

Although different developers might approach the application of the patterns differently, depending on their experience, the following guidelines can be adopted:

(1) *Identify the secure goals of the system.* In this step, the secure goals of the system as derived during late requirements analysis. For instance, as shown in Figure 6, there are three main secure goals for the eSAP system: Ensure Data Availability, Ensure Data Integrity, and Ensure System Privacy.
(2) *Identify the secure tasks for each secure goal.* This step involves the identification of secure tasks that correspond to the secure goals identified during the previous step. Consider, as an example, the Ensure System Privacy secure goal of the eSAP system. This is achieved by the following secure tasks: Check Information Flow, Check Authentication, and Check Access Control.
(3) *Identify the security challenges introduced by the secure tasks.* During this step, the security challenges facing the developer trying to satisfy the secure tasks are identified. Consider, for instance, the Check Authentication and Check Access Control secure tasks identified in the previous step. In order to satisfy the former, all communications and information exchanges of the eSAP system with external agents must be authenticated, whereas to satisfy the latter, every agent in the system must only have access to designated resources.
(4) *Identify the patterns that address the security challenges.* When the security challenges have been identified, the pattern language can be used to provide solutions to those challenges. In the example, in response to the first challenge, the *Agency Guard* pattern can be applied to restrict access to the eSAP system as a whole, and *Agent Authenticator* to provide authentication checks.

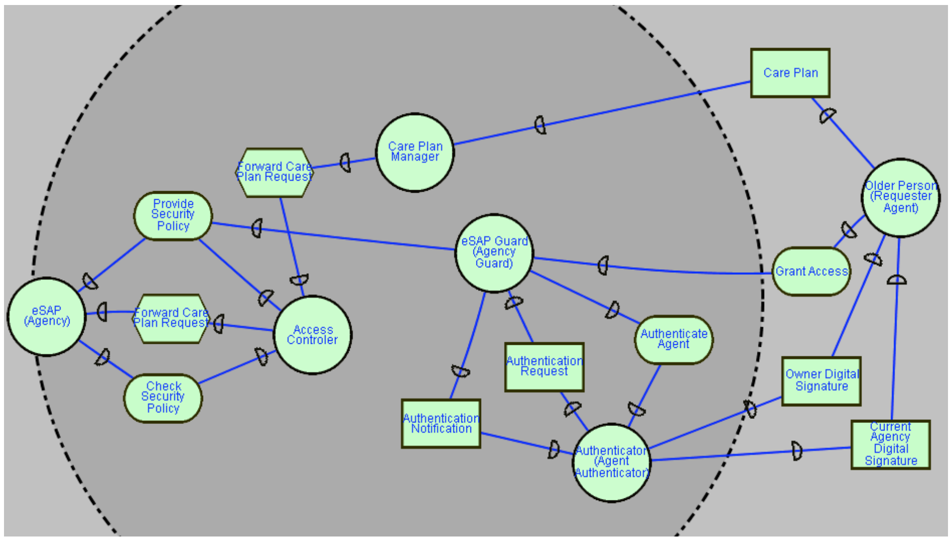
Care Plan
Forward Care Plan Request
Care Plan Manager
Older Person (Requester Agent)
Provide Security Policy
eSAP Guard (Agency Guard)
Grant Access
eSAP (Agency)
Forward Care Plan Request
Access Controler
Authenticate Agent
Authentication Request
Owner Digital Signature
Check Security Policy
Authentication Notification
Authenticator (Agent Authenticator)
Current Agency Digital Signature

Fig. 12. Example of applying the security patterns. This diagram is the result of applying the patterns *Agency Guard*, *Agent Authenticator*, and *Access Control* in sequence.

5.2. *Application to the eSAP Example*

The application of the security patterns amounts to instantiating the generic roles described in their solutions to concrete actors in the system under development. For example, consider the the Obtain Updated Care Plan Information secure goal of the Older Person actor modelled in Figures 4 and 5. The system needs to ensure that only entitled users can obtain access to the plan information.

We start by applying the *Agency Guard* pattern, which restricts access to the agency to a single point. When we instantiate the generic roles prescribed by this pattern, Older Person takes on the role of Requester Agent, the eSAP system corresponds to Agency, and a new actor, eSAP Guard, is introduced to assume the role of Agency Guard. The result is shown in Figure 12. As shown, the pattern also introduces two new dependencies (Grant Access and Provide Security Policy).

Next we apply the *Agent Authenticator* pattern to ensure the identity of the Older Person agent (Check Authentication subgoal of Ensure Data Privacy), and the *Access Controller* pattern in order to restrict the Older Person's access to their own medical records (Check Access Control subgoal of Ensure Data Privacy). The *Sandbox* pattern is not applicable, since the Older Person is a trusted user.

Application of these patterns leads to the introduction of more new actors and social dependencies between those actors and existing ones. The Authenticator fills the role of Agent Authenticator, and with it one new goal dependency (Authenticate Agent), and four resource dependencies (between Authenticator and eSAP Guard, and Older Person, respectively) are added to the model. Similarly, with the Access

Controller two new goals (Check Security Policy, and Provide Security Policy), as well as two new task dependencies (involving the eSAP Agency, the eSAP Agency Guard, and the Care Plan Manager in the role of the Resource Manager) are introduced.

The use of the patterns of our pattern language helps developers to delegate the responsibilities of particular security goals to particular actors defined by the patterns. In addition, the developer knows the consequences that each pattern introduces to the system under development. For example, the application of *Agent Authenticator* means that during implementation only the Agent Authenticator must be checked for assurance, while the application of *Access Controller* implies that different policies could be used for accessing different resources.

6. Assessing Completeness of the Pattern Language

In this section we address the issue of the completeness of our language by formalizing properties of our patterns. The basic idea is to follow the "uses" links between patterns, which can be found in the Related Patterns sections, and record the problems addressed by each pattern, as well as the new problems they raise. From this we can either conclude that the application of our patterns helps establish security (that is, that all security problems raised are resolved), or that we need to add more patterns to our language in order to resolve the open problems.

The *properties* of a pattern that we need to formalize are, therefore, problem, solution, and consequences (new problems raised). Problems and solutions will be represented as Formal Tropos properties. Formal Tropos (FT) is a a specification language that offers all the standard mentalistic notions of Tropos.[14] It supplements them with a rich temporal specification language inspired by KAOS.[10] FT allows for the description of the dynamic aspects of Tropos models. More precisely, in FT we focus not only on the intentional elements themselves, but also on the circumstances in which they arise, and on the conditions that lead to their fulfillment.

A FT specification describes the relevant elements (actors, goals, dependencies, etc.) of a domain and their relationships. The description of each of the elements is structured in two layers: an outer and an inner layer. The outer layer is similar to a class declaration. It associates a set of attributes with each element that define its structure. There is also a set of predefined special attributes such as **Actor**, **Depender** and **Dependee**. The inner layer expresses constraints on the lifetime of the objects, given in a typed first-order linear-time temporal logic.

For reasons of space, we only present the formalization of patterns related to authentication and their relationships. For each pattern, we formalize the problem addressed, the solution, and the new problems introduced. The formalization of the problems appear where they are first raised, and are referenced in later patterns. This approach also proved helpful in ensuring that the description of a problem did not use any of the new intentional elements introduced by the solution.

The following is an excerpt of the initial outer layer of the FT specification that we will refer to in the formalization of pattern properties. For each pattern we will indicate how this specification is refined (by specifying the new elements and attributes added by the pattern) upon applying the pattern.

Actor Agency

Actor RequesterAgent
  Attribute
    constant owner : AgentOwner

Actor AgentOwner

Goal Dependency GainAccessToAgency
  Depender RequesterAgent
  Mode achieve

Resource Dependency AccessRequest
  Dependee RequesterAgent
  Attribute
    constant resource : Resource
  Mode achieve

Resource Dependency Resource
  Dependee RequesterAgent

6.1. *Agency Guard*

The formalization of *Agency Guard* models the problem that requester agents can access the agency from multiple places via the GainAccessToAgency goal dependency. Problem P1 specifies that there is a way for a RequesterAgent to gain access to the agency by exploiting multiple GainAccessToAgency dependencies in which it participates. Solution S1 resolves this problem, as specified in the last clause of the assertion. However, it also adds a new problem (P2). The formalization of problem P2 states that ensuring that agents can only access the agency through a single point *does not* also ensure that the agents are who they claim to be.

Problem

/ P1: A malicious agent can gain unauthorized access to the agency from multiple places, not all of which provide the same level of security. */*

$$\exists \mathsf{ra} : \mathsf{RequesterAgent}\ (\exists \mathsf{ga1}, \mathsf{ga2} : \mathsf{GainAccessToAgency}\ (\mathsf{ga1}.\textbf{depender} = \mathsf{ra} \wedge \mathsf{ga2}.\textbf{depender} = \mathsf{ra} \wedge \mathsf{ga1}.\textbf{dependee} \neq \mathsf{ga2}.\textbf{dependee})))$$

Solution

/ S1: Ensure that there is only a single point of access to the agency. */*

$\forall$ ra : RequesterAgent ($\forall$ ga1, ga2 : GainAccessToAgency (ga1.**depender** = ra $\wedge$
ga2.**depender** = ra) $\rightarrow$ (ga1.**dependee** = ga2.**dependee**))

As part of the solution the outer layer model is refined by adding the following new elements (one actor and two dependencies) and new attributes:

Actor AgencyGuard */* added */*

Goal Dependency GainAccessToAgency
  Depender RequesterAgent
  Dependee AgencyGuard */* added */*
  Mode achieve

Resource Dependency AccessRequest
  Depender AgencyGuard */* added */*
  Dependee RequesterAgent
  Attribute
    constant resource : Resource
  Mode achieve

Resource Dependency SecurityPolicy */* added */*
  Depender AgencyGuard
  Dependee Agency
  Mode achieve

Goal Dependency GrantDenyAccess */* added */*
  Depender Agency
  Dependee AgencyGuard
  Mode achieve

New problems

/ P2: Agents can enter the agency by posing as another agent. */*

$\exists$ ar : AccessRequest ($\exists$ ra : RequesterAgent (ar.**dependee** = ra $\wedge$
ar.**dependee**.owner $\neq$ ra.owner))

6.2. *Agent Authenticator*

The solution (S2) of *Agent Authenticator* resolves problem P2. It states that if RequesterAgents are signed with their owners' private keys, they can be authenticated via the corresponding public keys, and can no longer masquerade as another agent.

Problem

/ P2: Agents can enter the agency by posing as another agent. */*

Solution

/ S2: Agents must prove their identity. Agents are authenticated via their owner's or their originating agency's public keys. */*

```
∀ ar : AccessRequest (∀ ra : RequesterAgent (ar.dependee = ra ∧
  ∀ ao : AgentOwner (ra.owner = ao ∧ ∃ ds : DigitalSignature (
    ds.dependee = ra ∧ ra.key = ao.privateKey)) →
    ar.dependee.owner = ra.owner)) /* via the owner's public key */
```

As part of the solution the outer layer model is refined as follows:

```
Actor AgentAuthenticator /* added */

Actor AgentOwner
  Attribute
    constant privateKey : Key /* added */

Actor RequesterAgent
  Attribute
    constant owner : AgentOwner
    constant key : Key /* added */

Resource Dependency Key /* added */
  Depender RequesterAgent
  Dependee AgentOwner
  Mode achieve

Resource Dependency DigitalSignature /* added */
  Depender AgentAuthenticator
  Dependee RequesterAgent
  Mode achieve

Resource Dependency AuthenticationDecision /* added */
  Depender AgencyGuard
  Dependee AgentAuthenticator
  Mode achieve

Resource Dependency AuthenticationRequest /* added */
  Depender AgentAuthenticator
  Dependee AgencyGuard
  Mode achieve
```

Note that, for reasons of space, authentication via trusted agencies was left out from this formalization. However, it would be included in a similar manner.

New problems

/ None */*

6.3. *Benefits of the Formalization*

The formalization allows us to model how the application of a given pattern results in assertions being added to the model. We can now formally reason about the security problems resolved by a given security solution. For example, consider the assertion made by solution S2. It states that the apparent initiator of an AccessRequest (that is, the agent owner on whose behalf the request is made) is identical with the owner of the RequesterAgent, if the agent has been signed with the initiator's private key. Thus, the application of the *Agent Authenticator* pattern eliminates the possibility of one agent masquerading as another (formalized as problem P2).

Formalization also leads to a deeper understanding of the patterns and their interrelationships, and confidence in the completeness of the pattern language. As a result we were able to discover problems with a given solution that were, in many cases, non-obvious observations about the solution. We were then also able to propose missing patterns to resolve those problems. For example, an implicit assumption made by solution S1 in the *Agency Guard* pattern was that the owner of the RequesterAgent and the agent owner on whose behalf the AccessRequest was made were one and the same. However, this assumption is not necessarily true, since the RequesterAgent can pose as another agent (as stated by problem P2).

In summary, the formalization allows us to decide when the application of our patterns resolves all security problems raised either at the late requirements analysis stage (which provide the initial input to the application of the pattern language), or when new problems are added as a result of applying patterns, and thus more patterns need to be applied, in turn, to resolve those problems.

7. Related Work

Our approach integrates two well known areas of software engineering research, agent-oriented software engineering and security patterns.

7.1. *Agent-Oriented Software Engineering*

Liu *et al.* have presented work to identify security requirements using agent-oriented concepts, whereas Yu and Cysneiros provide an approach to model and reason about non-functional requirements (with emphasis on privacy and security).[20,30] In his Non-functional Requirements (NFR) framework, Chung applies a process-oriented approach to represent security requirements as potentially conflicting goals, and explains how they can be used during the development of software systems.[8] In addition, Jürgens proposes UMLsec, an extension of the Unified Modeling Language (UML), to include modelling of security related features, such as confidentiality and access control.[17] The concept of obstacles is used in the KAOS framework to capture undesirable properties of the system, and to define and relate security requirements to other system requirements.[10] Sindre and Opdahl define the concept of a miuse case to describe security-related functions that a system should not allow.[25]

The main problem with these approaches is that they only provide solutions to isolated problems. that is they consider security a one-dimensional problem. For instance, the NFR framework assumes that all developers have some kind of security knowledge. UMLsec only supports the design phase, whereas the work of Liu *et al.* is focused on the requirements elicitation stage. By contrast, our approach considers all development stages from early requirements elicitation to design.

7.2. *Security Patterns*

The idea of developing a set of patterns or a pattern language for capturing proven security solutions is, by itself, not new. Yoder and Barcalow proposed a set of patterns that can be applied when adding security to an application.[27] Lee Brown *et al.* proposed an Authenticator pattern, which performs the authentication of a requesting process before granting access to distributed objects.[19] Building on this work, Fernandez and Pan document a pattern language for security models.[13] Finally, Schumacher applies the pattern approach to the security problem by proposing a set of patterns, which contribute to the overal process of security engineering.[24]

Although this review is by no means complete, most of the proposed security-related patterns and pattern languages have been developed from an object-oriented perspective. As stated by Fernandez and Pan, the intent of these patterns is to "specify the accepted models as object-oriented patterns".[13] However, it has been argued that there is no single paradigm or language for implementing patterns. Patterns can be integrated with any paradigm used for constructing software systems. We believe that the introduction of the agent-oriented paradigm has opened another important area for the use of patterns, as discussed next.

7.3. *Merging these Areas*

One of the main arguments for the use of an agent-oriented, as opposed to an object-oriented approach, is that agent-oriented concepts for the decomposition of a system, such as goals, plans and actors, are more intuitive and easier to use than object-oriented concepts, such as data, behaviour and objects.[6,16] In addition, the agent approach provides a paradigm for designing the types of complex distributed systems that require, among other things, a common abstraction for representing both human users (via agents that act on their behalf), as well as software entities that manage other subsystems or resources (agents as resource managers). Thus, the combination of patterns and agent-orientation is likely to be very important, since the higher level of abstraction and the encapsulation of agents allows an easier identification and characterisationof reusable parts.

However, if patterns and pattern languages are to be integrated with agent-oriented approaches, it is necessary to develop them using agent-oriented concepts, such as intentionality, dependency, autonomy, sociality and identity.[28] In doing so, we feel it is essential to describe the structure of a pattern not only in terms of the messages exchanged between the participating agents, but also in terms of their social dependencies and intentional attributes, such as goals and tasks.

In this way, we can achieve a complete understanding of the social and intentional dimensions of a pattern, two factors of greate importance when developing information systems from an agent-oriented perspective. This view has been also argued by other researchers. For example, Deugo *et al.* conclude that differences in the way agents and objects communicate and interact with their environment, the level of autonomy agents possess, and the fact that agents are often highly mobile motivate the separate notion of an agent pattern.[11]

Thus, research on agent patterns has started to evolve and some catalogues of agent patterns have already been presented in the literature. Hayden *et al.*, provide a catalogue of coordination patterns inherent in multi-agent architectures.[15] The proposed patterns are grouped into four basic architectural styles: hierarchical, federated, peer-to-peer, and agent-pair. Kendall *et al.* propose patterns for intelligent and mobile agents.[18] In this work the authors argue that agent systems must have a strong foundation based on well defined patterns. Aridor and Lange present a catalogue of design patterns for creating mobile agent applications.[4] They divide their patterns into three classes: travelling, task and interaction.

Although these approaches are helpful and provide a first step for the integration of patterns and agent-oriented software engineering, they also demonstrate two important limitations. Firstly, there is a lack of a framework to support the analysis of the requirements, and determine precisely the context within which a pattern can be applied. Secondly, and more specific to our topic, there is a lack of agent-oriented security patterns. Our work provides a step towards overcoming these limitations by defining an agent-oriented security pattern language, and by integrating this language with an agent-oriented software engineering methodology.

8. Conclusion

In this paper we have proposed an approach for the development of secure information systems that merges two important software engineering paradigms: agent-oriented software engineering (AOSE) and patterns. In particular, we have described a pattern language based on agent-oriented concepts, and we have demonstrated its applicability, with the aid of a real-case study, and how it can be integrated within the architectural design stage of the Tropos agent-oriented methodology. Moreover, we have addressed the completeness of the patterns of our language by formalising the properties of its patterns. We believe that the integration of these two paradigms provides a complete and mature solution for the development of secure information systems. We consider the integration to be *complete*, since we believe that those paradigms complement each other:

- AOSE provides concepts and notations suitable for modeling security issues in information systems, such as autonomy, intentionality and sociality.
- Patterns complement agent-oriented techniques by transferring security knowledge to non-security application experts in an efficient manner.

We also consider the integration to be *mature* as such an integration will make security solutions more widely available, providing novice and non-security expert developers with the capability of implementing secure information systems.

Future work includes extending our pattern language with more security patterns, and identifying its relationship to other (non-security) agent-based patterns. In addition, we will work towards the integration of the pattern language with other development phases of the Tropos methodology.

References

1. C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Constructions*, Oxford University Press, 1977.
2. E. Amoroso, *Fundamentals of Computer Security Technology*, Prentice Hall, 1994.
3. R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, Wiley, 2001.
4. Y. Aridor, D. Lange,Agent Design Pattern: Elements of Agent Application design, in *Proceedings of the International Conference on Autonomous Agents*, ACM Press, 1998.
5. K. Beck, and R. Johnson, Patterns Generate Architectures, in: *European Conference on Object-Oriented Programming (ECOOP)*, LNCS 821, 139–149, Springer, 1994.
6. P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini, TROPOS: An Agent-Oriented Software Development Methodology, *Journal of Autonomous Agents and Multi-Agent Systems*, 8(3), 203–236, Kluwer, 2004.
7. L. Chung, Dealing with Security Requirements During the Development of Information Systems, in: *Conference on Advanced Information Systems (CAiSE)*, 234–251, ACM, 1993.
8. L. Chung, B. Nixon, E. Yu, and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*, Kluwer, 2000.
9. J. Coplien, *Software Patterns*, SIGS books, 1996, also available from: `http://users.rcn.com/jcoplien/Patterns/WhitePaper/SoftwarePatterns.pdf`.
10. A. Dardenne, A. van Lamsweerde, S. Fickas, Goal-Directed Requirements Acquisition, *Science of Computer Programming*, special issue on the *6th International Workshop on Software Specification and Design*, Vol. 20, 3–50, 1993.
11. D. Deugo, M. Weiss, E. Kendall, Reusable Patterns for Agent Coordination, In *Coordination of Internet Agents*, Springer, 2001.
12. P. Devanbu, and S. Stubblebine, Software Engineering for Security: a Roadmap, in *Conference on the Future of Software Engineering*, 226–239, ACM, 2000.
13. E. Fernandez, and R. Pan, A Pattern Language for Security Models, in: *Conference on Pattern Languages of Programs (PLoP)*, 2001.
14. A. Fuxman, L. Liu, *et al.*, Specifying and Analyzing Early Requirements in Tropos, *Journal of Requirements Engineering*, **9**:2. 132–150, May 2004.

15. S. C. Hayden, C. Carrick, Q. Yang, A Catalog of Agent Coordination Patterns,*Agents 1999*,pp 412-413
16. N.R. Jennings, An Agent Based Approach for Building Complex Software Systems, *Communications of ACM*, **44**:4, April 2001.
17. J. Jürjens, Towards Development of Secure Systems with UMLsec, in: *Conference on Fundamental Approaches to Software Engineering (FASE)*, LNCS 2029, 187–200, Springer, 2001.
18. A. Kendall, P.V. Murali Krishna, C. V. Pathak, C.B. Suresh, Patterns of Intelligent and Mobile Agents, *Proceedings of the 2nd International Conference on Autonomous Agents*, 1998
19. F. Lee Brown, E. Fernandez, The Authenticator Pattern, in: *Conference on Pattern Languages of Programs (PLoP)*, 1999.
20. L. Liu, E. Yu, J. Mylopoulos, Analysing Security Requirements As Relationships Among Strategic Actors, in: *Second Symposium on Requirements Engineering for Information Security (SREIS)*, 2002.
21. H. Mouratidis, P. Giorgini, G. Manson, and I. Philp, A Natural Extension of Tropos Methodology for Modelling Security, in: *Agent-Oriented Methodologies Workshop* at OOPSLA, 2002.
22. H. Mouratidis, P. Giorgini, and G. Manson, Modelling Secure Multiagent Systems, in: *International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 859–866, ACM, 2003.
23. H. Mouratidis, I. Philp, and G. Manson, Analysis and Design of eSAP: An Integrated Health and Social Care Information System, in: *International Symposium on Health Information Managements Research (ISHIMR)*, 2002.
24. M. Schumacher, *Security Engineering with Patterns*, LNCS 2754, Springer, 2003.
25. G. Sindre, and A. Opdahl, Eliciting Security Requirements by Misuse Cases, in: *Conference on Technology of Object-Oriented Languages and Systems (TOOLS)*, 120–131, 2000.
26. J. Viega and G. McGraw, *Building Secure Software-How to Avoid Security Problems the Right Way*, Addison-Wesley, September 2002.
27. J. Yoder, and J. Barcalow, Architectural Patterns for Enabling Application Security, in: *Conference on Pattern Language of Programs (PLoP)*, 1997.
28. E. Yu., *Modelling Strategic Relationships for Process Reengineering*, Ph.D. thesis, Department of Computer Science, University of Toronto, Canada, 1995.
29. E. Yu, Agent-Oriented Modelling: Software versus the World, in: *International Workshop on Agent-Oriented Software Engineering (AOSE)*, LNCS 2222, 206–225, Springer, 2002.
30. E. Yu, L. Cysneiros, Designing for Privacy and Other Competing Requirements, in: *Symposium on Requirements Engineering for Information Security (SREIS)*, 2002.