Security Patterns Meet Agent Oriented Software Engineering: A Complementary Solution for Developing Secure Information Systems

Haralambos Mouratidis[1], Michael Weiss[2], and Paolo Giorgini[3]

[1] School of Computing and Technology, University of East London, England
h.mouratidis@uel.ac.uk
[2] Dept. of Computer Science, Carleton University, Ottawa, Canada
weiss@scs.carleton.ca
[3] Dept. of Information and Communication Technology, University of Trento, Italy
paolo.giorgini@dit.unitn.it

Abstract. Agent Oriented Software Engineering and security patterns have been proposed as suitable paradigms for the development of secure information systems. However, so far, the proposed solutions are focused on one of these paradigms. In this paper we propose an agent oriented security pattern language and we discuss how it can be used together with the Tropos methodology to develop secure information systems. We also present a formalisation of our pattern language using Formal Tropos. This allows us to gain a deeper understanding of the patterns and their relationships, and thus to assess the completeness of the language.

1 Introduction

Information systems security is definitely not a new topic, since its history starts in the sixties [12]. Nevertheless, only recently more importance has been given to information security, and it is considered now one of the main issues during information systems development. This situation is the result of two main factors: (1) the wide usage of information systems by institutions, companies and individuals, and, therefore, the storage of important information; and (2) the increasing number of information systems security criminals such as hackers and attackers. Research on the security of information systems has mainly focused on the definition of security protocols, security mechanisms and other technical solutions. Yet, it has been widely argued over the last few years that security is not simply a technical issue, and that security solutions cannot be blindly inserted into information systems, but security considerations need to be tightly integrated with the development of information systems [14,4,10].

Following this argument, two software engineering paradigms, namely agent oriented software engineering and security patterns, have been proposed (e.g., in [8] and [13]) as promising paradigms for the development of secure information systems. On the one hand, it has been argued [10] that agent oriented software

L. Delcambre et al. (Eds.): ER 2005, LNCS 3716, pp. 225–240, 2005.

engineering is one of the most natural ways of characterising security issues in information systems, since characteristics, such as autonomy, intentionality and sociality, provided by the use of agent orientation, allow developers first to model the security requirements in high-level, and then incrementally transform these requirements to security mechanisms. On the other hand, 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]. In addition, security patterns introduce several layers of abstraction and thus help to close the gap between security specialists and software engineers.

So far, the solutions proposed by these two paradigms are divided; that is, they only consider either an agent oriented or a security pattern solution. We believe that the integration of agent oriented software engineering and security patterns represents an effective solution for the consideration of security issues during the development stages of information systems. This is mainly due to the appropriateness of an agent oriented philosophy for dealing with the security issues that exist in a computer system and the appropriateness of patterns to transfer security related knowledge to non-security specialists.

Secure Tropos [10] extends the agent oriented software engineering methodology Tropos [3] by providing a set of security-related concepts and processes to allow developers to consider security issues throughout the development stages. Secure Tropos supports three development stages: the early requirements analysis stage, in which the social issues related to the security of the system are identified and analysed; the late requirements analysis stage, in which the technical issues related to the security of the system are identified and analysed; and the architectural design stage, in which the architectural style of the system is defined with respect to the system's security requirements and the requirements are transformed into a design. However the latter could be a very difficult task, especially for a developer without knowledge of security, possibly resulting in the development of a non-secure system. For this reason, we propose to complement secure Tropos with the use of security patterns. Security patterns capture existing proven experience about how to deal with security problems during the software development and they help to promote best design practices.

Building on our previous work [9,15] we introduce an approach for modelling security issues in information systems using agent-oriented software engineering and security patterns. Section 2 introduces the proposed security pattern language. Section 3 discusses the formalisation of the pattern language. Section 4 describes how it can be applied, and Section 5 concludes this paper.

2 Security Pattern Language

Patterns by themselves are only point solutions, and they are usually organised into pattern languages. A *pattern language* is a set of closely related patterns that guide the developer through the process of designing a system [1]. As the patterns from a pattern language are applied, each pattern suggests new patterns to be applied that further refine the design, until no more patterns can be applied.

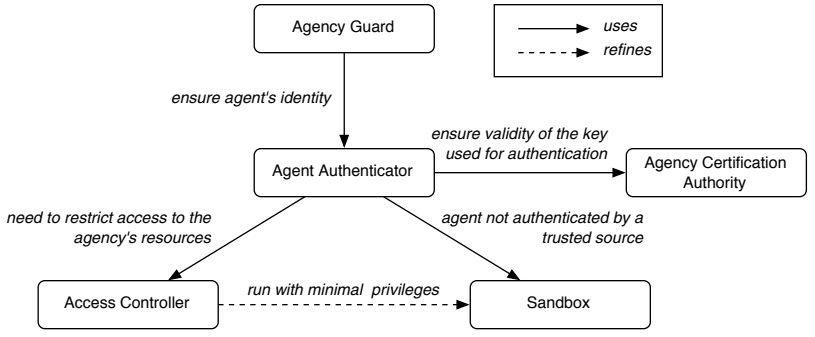

Fig. 1. Roadmap of the security pattern language

Since our aim is to integrate agent oriented software engineering and security patterns, the pattern language should employ agent-oriented concepts, such as intentionality, autonomy, sociality, and identity. Therefore, 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.

We use the so-called Alexandrian format for organising each pattern [1]. The sections of a pattern are *context*, *problem*, *solution*, and *consequences*. Brief descriptions of the problem and solution are put in boldface, followed by more detailed discussions. The consequences are organised into *benefits*, *liabilities*, and *related patterns*. Figure 1 provides a roadmap of our pattern language. The directed links show dependencies between patterns, and point from a pattern to the patterns that developers may want to consult next. It should also be stressed that the patterns of the language have been identified from real implementations of agent-based systems, and their initial versions have been workshopped at a patterns conference [9] in order to validate and improve them.

2.1 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 unauthorised 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 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

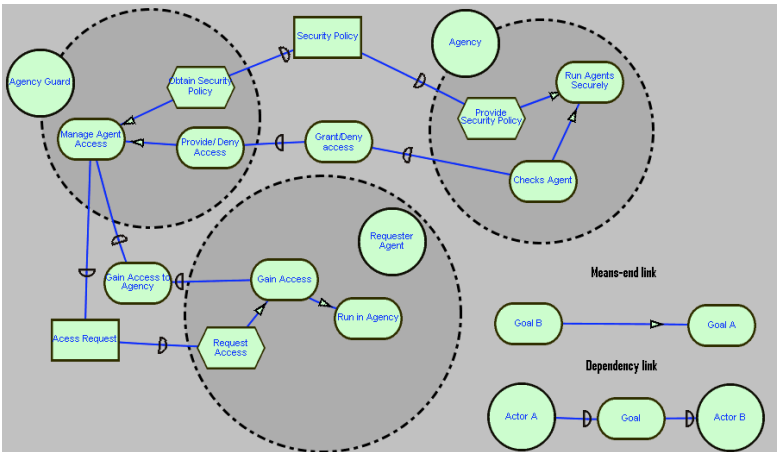

Fig. 2. Structure of the *Agency Guard* pattern

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 gain access to an Agency (either to access resources or move to this agency) its requests must be directed to an Agency Guard, which grants or denies access requests according to the agency's security policy. The Agency Guard is the only point of access to the Agency, and cannot be bypassed, meaning all access requests must go through it. In traditional terms, the concept of an Agency Guard is referred to as a monitor [2].

The structure of the pattern in terms of the actors involved and their social dependencies is shown in Figure 2 using the Tropos notation. Each circle represents an actor, and each dependency link between two actors indicates that one actor depends on the other for some goal (oval), task (hexagon) or resource (rectangle) to be achieved. Moreover, actors are analysed internally (internal analysis is indicated within the dashed line circles) with the aid of means-end links, which are used to indicate (alternative) means (goals/tasks) for reaching a goal. For example, the Agency depends on the Agency Guard to Grant/Deny Access to the Agency according to the Agency's security policy.

***

Benefits:

- It is easier to secure a single point of access, rather than many backdoors.
- 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:

- Requester Agents need to present all their credentials (including identity), although they may only be required for some operations on the agency.
- A malicious Requester Agent may masquerade its identity.
- A single point of access to the agency can degrade the performance of the agency (that is, its response time for handling access requests).
- The Agency Guard is a single point of failure. If the Agency Guard fails, the security of the agency as a whole is at risk.
- We cannot prevent Requester Agents from attempting to circumvent the Agency Guard. We, therefore, also need to log access requests.

Related patterns:

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

2.2 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:

Authenticate agents as they enter the agency.

Requester Agents must be authenticated by the Agency. By authenticating the agent, the Agency Guard ensures the agent comes from an owner trusted by the Agency. Each Requester Agent's owner and Agency have a pair of public/private keys. The Agent Authenticator can authenticate the Requester Agent in two ways: the agent can be digitally signed with its 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 execution privileges.

The structure of the pattern is shown in Figure 3.

***

Benefits:

- Since authentication concerns are dealt with in a single location, it is not necessary to provide each agent with its own authentication mechanism.

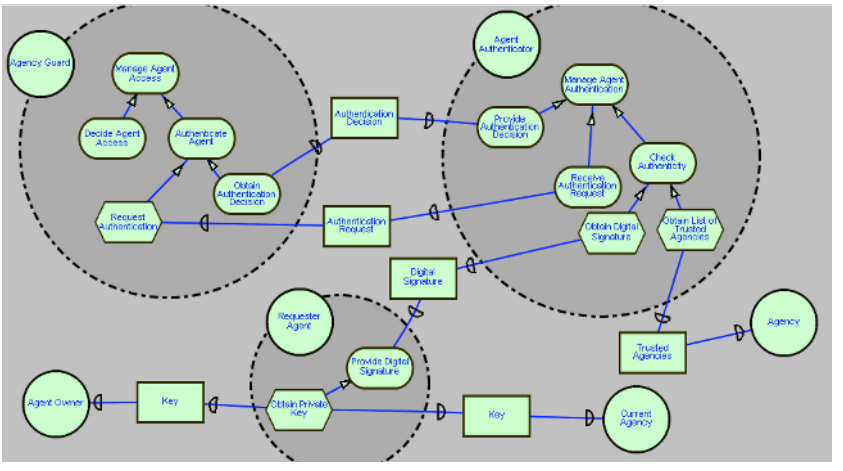

Fig. 3. Structure of the *Agent Authenticator* pattern

– 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 verified for correct enforcement of the agency's authentication 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.
– The public key used to authenticate the Requester Agent may be invalid.
– This solution may be too restrictive, as it prevents agents that provide services that the agency cannot provide itself, but cannot be authenticated, from executing.

Related patterns:

– *Access Controller* – restricts access to the agency's resources.
– *Access Certification Authority* – ensures validity of the public key used to authenticate the Requester Agent.
– *Sandbox* – allows running an unauthenticated agent with minimal privileges.

2.3 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.

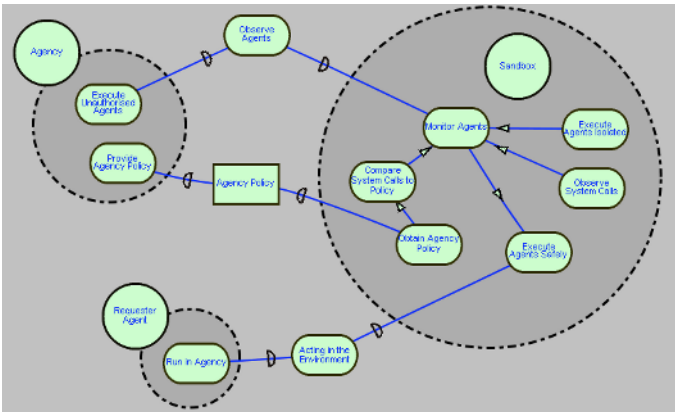

Fig. 4. Structure of the *Sandbox* pattern

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 minimise 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.

This solution prevents malicious agents from performing unauthorised operations. The agent is allowed to destroy anything within a restricted environment (a Sandbox), but 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 agent, and compares them to the agency's policy. If any violations occur, the Agency can shut down the suspicious agent.

The structure of the pattern is shown in Figure 4.

***

Benefits:

- Agents not authorised but, nonetheless, 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), and prevent them from occurring.

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.

No related patterns.

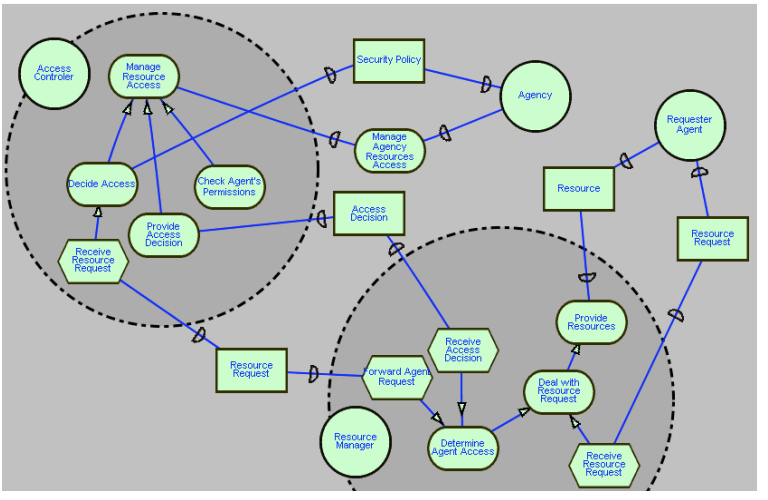

Fig. 5. Structure of the *Access Controller* pattern

2.4 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 be able to access only specific resources.

***

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. 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. When a Requester Agent requests access to a resource, the request is directed to the Access Controller, which then checks the security policy and decides whether the access request should be approved or rejected. The access decision is then forwarded to the corresponding Resource Manager.

The structure of the pattern is shown in Figure 5.

***

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.

No related patterns.

2.5 Agent Certification Authority

... you are using *Agent Authenticator* to authenticate agents coming to the agency. Each agent has a certificate, which contains a public key. The agent proves its identity by signing with its private key, which might come from the agent's owner or the agency that it currently resides. However, such a signature is only valid, if the corresponding public key has been certified.

***

Malicious agents wishing to access the agency might try to authenticate using invalid keys or keys that are not certified.

The agency should not authenticate such agents since they might endanger the security of the agency.

Therefore:

Authenticate agents only if their public key has been certified by a trusted certificate authority.

An Agent Certification Authority is used to certify the agent's public key. The Requester Agent sends a certification request to the Agent Certification Authority. When the Agent Certification Authority receives the request, it verifies its validity by checking when the request was generated, that the signed message can be verified using the Requester Agent's public key, and that the public key of the agent is not already in use. If the request is valid, the Agent Certification Authority generates a signed certificate that binds the public key to the agent. Before actually sending the certificate to the agent, the Agent Certification Authority creates an entry in its certification database. When the agent receives the certificate, it can prove its identity by signing with its private key.

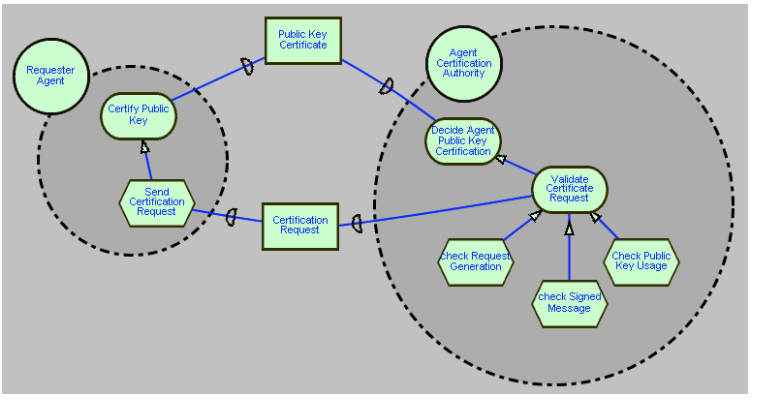

Fig. 6. Structure of the *Agent Certification Authority* pattern

The structure of the pattern is shown in Figure 6.

***

Benefits:

- The validity of the Requester Agent's public key is verified
- The Requester Agent's claim that a public key belongs to its owner or to its originating agency is verified.

Liabilities:

- Scalability is limited when too many Requester Agents try to obtain a verification of their public keys at once. This can be solved by having more than one Agent Certification Authority. Different Agent Certification Authorities are aware of each other and can route certificates between them if necessary.

No related patterns.

3 Formalising the Pattern Language

An important consideration in the development of a pattern language is to assess its completeness. For this reason, in addition to the graphical representation, we have developed a formalisation of the language. Patterns are formalized in terms of the problems they address, and the solutions they offer. We consider a pattern language to be *complete*, if the solutions proposed by the patterns contained in the language address all the problems raised. It is important to note that completeness can only be assessed with regard to a stated set of problems. As new security problems are identified, the pattern language needs to be extended.

As the roadmap in Figure 1 illustrates, the patterns in a pattern language are interconnected. The links indicate uses and refines relationships as defined in [11]. Intuitively, uses can be interpreted in either one of two ways. Either a particular problem is not addressed by a pattern, and the problem has to be resolved by another pattern in the pattern language; or the application of a pattern raises new problems that must be addressed by further patterns in the pattern language. The basic idea underlying our formalization is to follow the uses links between the patterns, and to 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. For a formal definition of the uses relationship, and a proof that this approach allows us to show the completeness of a pattern language see [13].

Our formalization of patterns is only a partial formalization. A full formalization may not even be desirable, since patterns are meant to be human-readable artifacts, and applying a pattern often requires adapting the pattern to the specific needs of a given context [1]. We, therefore, focus on the following sections of a pattern: problem (addressed by this pattern), solution (how the problem is addressed), and consequences (new problems raised). The solution is included

in the formalization, since the application of a pattern results in elements being added to the model, which other problem statements may refer to.

We use Formal Tropos (FT) [7] to describe problems and solutions. 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 into 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 **depender** and **dependee**. The inner layer expresses constraints on the lifetime of the objects, given in a typed first-order linear-time temporal logic.

In passing, it should be noted that a solution that establishes security does not necessarily imply that it is the best solution in terms of other system qualities. Not included in our formalization are non-security softgoals such as complexity. The contributions to non-security softgoals could be used to compare alternative selections of patterns in terms of the quality of the overall solution (i.e., the combined result of applying the patterns). We will incorporate the formalization of non-security softgoals in our future work.

For reasons of space, we present only the formalization of the patterns related to authentication and their relationships. However, the same principles can be used to formalise the rest of the patterns. We present the problem addressed, solution, and new problems introduced by each pattern. The formalization of problems appear where they are first raised, and are referenced in later patterns. Such an approach also proved helpful in ensuring that the description of problems does not make use of any of the new model elements introduced by the solution, but which were not part of the model before the pattern was applied.

To represent problems and solutions in FT we express them using global assertions. These assertions are first-order predicate expressions over model components. Problems are thus statements about the current model. If the assertion holds true, the pattern is applicable. After applying the pattern, the solution statement is asserted, possibly introducing new model elements, and the assertion for the problem no longer holds. The new problems raised by a pattern are assertions that enable the application of further patterns, until no more new problems are raised. The three patterns whose formalization we will present are: *Agency Guard*, *Agent Authenticator*, and *Agent Certification Authority*.

3.1 Agency Guard

This pattern states that RequesterAgents can access the agency from multiple places via the GainAccessToAgency goal dependency. The formalization of the problem (P1) specifies that a RequesterAgent can 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. Problem P2 also introduces the notion of the owner of a RequesterAgent. In essence, the formalization of P2 indicates that just ensuring that agents can only access the agency through a single point does not ensure that the agents are who they claim to be. This problem needs to be addressed separately.

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

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

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

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

3.2 Agent Authenticator

The pattern resolves problem P2. Solution S2 states that RequesterAgents signed with the private keys of their owners (the DigitalSignature) can be authenticated via the corresponding public keys. Thus, they can no longer masquerade as another agent. However, this solution hinges on the fact that the agency knows the valid public key of the RequesterAgent's owner. But this is generally not the case, as described by problem P3. In fact, a malicious agent may claim that its owner is ra.owner = ao1, whereas, it is ar.**dependee**.owner = ao2. The formalization introduces two new attributes: the key attribute of a RequesterAgent, and the privateKey attribute to be associated with RequesterAgent owners.

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

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

$\forall$ ar : AccessRequest ($\forall$ ra : RequesterAgent (ar.**dependee** = ra $\wedge$
$\forall$ ao : AgentOwner (ra.owner = ao $\wedge$
$\forall$ ds : DigitalSignature (ds.**dependee** = ra $\wedge$
ra.key = ao.privateKey $\rightarrow$ ar.**dependee**.owner = ra.owner))))

Consequences: */* P3: The agent's public key may not be valid or certified. A malicious agent can exploit this by signing with its own private key. */*

$\exists$ ar : AccessRequest ($\exists$ ra : RequesterAgent (ar.**dependee** = ra $\wedge$
$\exists$ ao1, ao2 : AgentOwner (ra.owner = ao1 $\wedge$
$\exists$ ds : DigitalSignature (ds.**dependee** = ra $\wedge$
ra.key = ao2.privateKey $\wedge$ ao1 $\neq$ao2 $\wedge$ar.**dependee**.owner = ao2))))

3.3 Agent Certification Authority

It is important to note that problem P3 is only stated in terms of the concepts used in the *Agent Authenticator* pattern. *Agent Certification Authority* adds to

these the concept of a PublicKeyCertificate. A PublicKeyCertificate is signed by a trusted AgentCertificationAuthority. This proves that the publicKey of an agent is, in fact, that of the agent's owner. This publicKey can now be used to detect invalid digital signatures of other agents masquerading as the same owner. The application of this pattern does not introduce new security problems.

Problem: */* P3: The agent's public key may not be valid or certified. A malicious can exploit this by signing with its own private key. */*

Solution: */* S3: Authenticate agents only if their public key is certified. */*

$$\forall \text{ ra} : \text{RequesterAgent } (\forall \text{ ao} : \text{AgentOwner } (\text{ra.owner} = \text{ao} \wedge \forall \text{ aca} : \text{AgentCertificationAuthority } (\forall \text{ pkc} : \text{PublicKeyCertificate } (\text{pkc.}\mathbf{dependee} = \text{aca} \wedge \text{pkc.}\mathbf{depender} = \text{ra} \wedge \text{pkc.publicKey} = \text{ao.publicKey} \rightarrow \text{ra.publicKey} = \text{ao.publicKey}))))$$

3.4 Practical Value of the Formalisation

Although developers do not need to be aware of the formalisation when employing the proposed pattern language, its practical value cannot be underestimated. It allows us to assess the completeness of our pattern language with regard to its ability to establish security. We can observe that:

- Using the formalisation we show how the application of a given pattern results in assertions being added to the model. These allow us to formally reason about the security problems resolved by a given security solution.
- Formalisation leads to a deeper understanding of the patterns. We were able to discover non-obvious problems with a given security solution and to detect that there were patterns missing from the language to resolve them.

As an example of the former, consider the assertion made by solution S2 that the apparent initiator of an AccessRequest must equal the owner of the RequesterAgent, if the request has been signed with the initiator's private key. This eliminates the possibility of one agent masquerading as another, and is formalized as problem P2. As an example of the latter, an earlier version of the pattern language did not include *Agent Certification Authority*. This pattern was added as a means of dealing with invalid public keys, and problem P3 provides a formal justification for this extension of the pattern language.

4 Applying the Language

To make it easier to understand the practical application of the pattern language, we consider how the language was applied to the electronic Single Assessment (eSAP) system case study first introduced in [10]. The eSAP case study involves the development of an information system to support an integrated assessment of the health and social care needs of older people in England. Due to lack of space we cannot present the complete analysis here. The main secure goals of the

eSAP system are: Ensure System Privacy, Ensure Data Integrity, and Ensure Data Availability. These have been further decomposed [10] into secure tasks such as Check Access Control, Check Authentication, and Check Information Flow.

According to secure Tropos, transforming security requirements to design is not an easy task, and it becomes more difficult if attempted by developers without much knowledge of security, which should be considered the norm rather than the exception. For example, from the analysis of the eSAP system, it is concluded that authentication and access control checks (amongst others) must be performed in order for the system to satisfy the system's secure goal Ensure Data Privacy. The system should be able to authenticate any agents that send a request to access information of the system, and the system must control access to its resources. Therefore, the developer must identify the appropriate actors (and their dependencies) to fulfil the above-mentioned security goals.

The proposed security pattern language can greatly help with the identification of these actors without endangering the security of the system. *Agency Guard* suggests a way of managing access to the eSAP system. *Agent Authenticator* can be used to enforce the agency's security policy. *Agent Certification Authority* describes how to certify the public key of a requester agent. *Access Controller* can be applied to perform access control checks. *Sandbox* is not applicable to the eSAP system. Not only does application of the patterns satisfy the fulfilment of the goals, but it also guarantees the validity of the solution. To apply a pattern, the developer must carefully consider the problem to be solved, and the consequences that the application of each particular pattern will have on the system. These consequences may introduce new problems that need to be resolved by other patterns until no problems remain. Figure 7 shows a possible use of the above-mentioned patterns in the eSAP system with respect to the Obtain Care Plan Information goal of the Older Person.

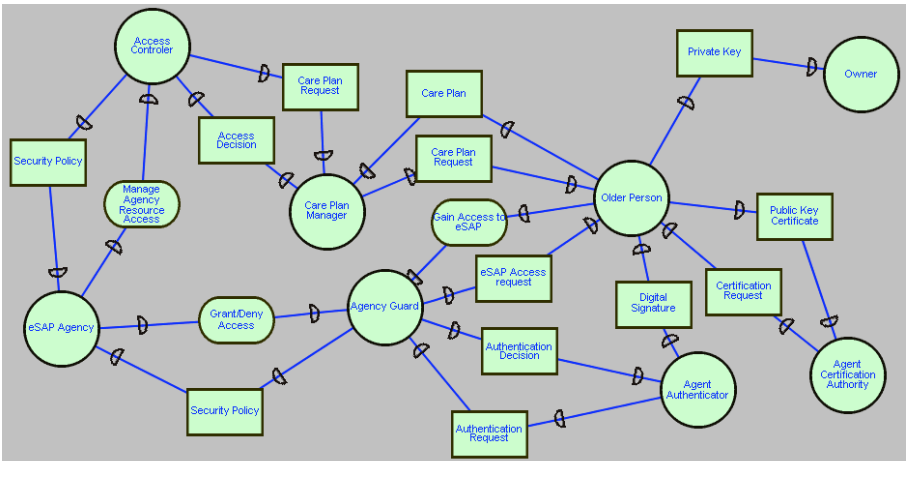

Fig. 7. Application of the patterns

We start by applying the *Agency Guard* pattern, which restricts access to the agency to a single point. As shown in Figure 7, the Older Person becomes the Requester Agent, the eSAP Agency corresponds to the Agency, and a new actor, the eSAP Guard, is introduced to assume the role of the Agency Guard. Next we apply the *Agent Authenticator* pattern to ensure the identity of the Older Person agent (the Check Authentication subgoal of Ensure Data Privacy), and the *Agent Certification Authority* pattern to ensure that the public key of the Older Person is certified. In addition, the Access Controller pattern is applied to restrict the Older Person's access only to their resources, i.e., to their own medical records. In this scenario, we assume that the Older Person should only be allowed to execute as an authorised user, and as such the *Sandbox* pattern is not applicable.

5 Conclusions

In this paper we propose an approach for the development of secure information systems that merges two important software engineering paradigms: agent oriented software engineering and security patterns. We believe this represents a suitable approach because agent orientation provides concepts such as autonomy, sociality and trust suitable for modelling security issues in information systems, whereas patterns complement agent orientation by transferring security knowledge to non security application experts in an efficient manner.

Approaches similar to ours have presented in literature. Liu et al. [8] have presented work to identify security requirements using agent oriented concepts. Jürgens proposes UMLsec [4], an extension of the Unified Modelling Language (UML), to include modelling of security related features, such as confidentiality and access control. The concept of an obstacle is introduced in the KAOS framework [5] to capture undesirable properties of a system, and to define and relate security requirements to other system requirements.

These approaches provide a first step towards the integration of security and software engineering and have been found helpful in modelling security requirements. However, they only guide the developer through how security can be handled within a certain stage of the development process. On the other hand, the area of security patterns is also very active. For example, Schumacher [13] applies the pattern approach to the security problem by proposing a set of patterns, called security patterns, which contribute to the overall process of security engineering; and Yoder and Barcalow [16] define architectural patterns for enabling application security. Fernandez and Pan [6] describe patterns for the most common security models. The main problem of these existing pattern languages is the lack of a framework to support the analysis of the security requirements and determine precisely the context in which a pattern can be applied.

By contrast, our approach merges the advantages of both the agent oriented and security patterns paradigms, by allowing developers to integrate a security pattern language within the development stages of an agent oriented software engineering methodology. This, in turn, allows developers to first analyse using agent oriented concepts the security issues related to the environment of the

system, and the system itself, identify a set of security requirements needed by the system, and transform these requirements to a design that satisfies them with the aid of security patterns. However, much more work is required, and we plan to extend our pattern language to include more patterns to address security-related issues such as the privacy of the agents' information.

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. 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*, Kluwer, 8(3), 203–236, 2004.
4. J. Jürjens, UMLsec: Extending UML for Secure Systems Development, *UML 2002*, LNCS 2460, 412-425, Springer, 2002.
5. A. Dardenne, A. van Lamsweerde and S. Fickas. Goal-directed Requirements Acquisition, *Science of Computer Programming*, Special issue on the 6th International Workshop of Software Specification and Design, 1991.
6. E. Fernandez and R. Pan. A Pattern Language for Security Models, *Conference on Patterns Languages of Programs* (PLoP), 2001.
7. A. Fuxman, *Formal Analysis of Early Requirements Specifications*, MSc thesis, University of Toronto, Canada, 2001.
8. L. Liu, E. Yu and J. Mylopoulos. Analyzing Security Requirements as Relationships Among Strategic Actors, *Symposium on Requirements Engineering for Information Security* (SREIS), 2002.
9. H. Mouratidis, P. Giorgini and M. Weiss. Integrating Patterns and Agent-Oriented Methodologies to Provide Better Solutions for the Development of Secure Agent Systems, Hot Topic on the Expressiveness of Pattern Languages, *ChiliPloP*, 2003.
10. H. Mouratidis, P. Giorgini and G. Manson. When Security meets Software Engineering: A Case of Modelling Secure Information Systems. *Information Systems* (in press).
11. J. Noble. Classifying Relationships between Object-Oriented Design Patterns, *Australian Software Engineering Conference* (ASWEC), 1998.
12. J. Saltzer and M. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), 1278-1308, September 1975.
13. M. Schumacher. *Security Engineering with Patterns*. LNCS 2754, Springer, 2003.
14. T. Tryfonas, E. Kiountouzis and A. Poulymenakou. Embedding Security Practices in Contemporary Information Systems Development Approaches, *Information Management & Computer Security*, 9(4), 183–197, 2001.
15. M. Weiss. Pattern Driven Design of Agent Systems: Approach and Case Study. *Conference on Advanced Information Systems Engineering* (CAiSE), LNCS 2681, Springer, 2003.
16. J. Yoder, J. Barcalow, Architectural Patterns for Enabling Application Security, *Conference on Pattern Languages of Programs* (PLoP), 1997.