Security Patterns: Comparing Modeling Approaches

Armstrong NHLABATSI[**]**, Arosha BANDARA**[**]**, Shinpei HAYASHI**[++]**, Charles B. Haley**[**]**, Jan JURJENS**[**]**, Haruhiko KAIYA**[+]**, Atsuto KUBO**[++]**, Robin LANEY**[**]**, Haris MOURATIDIS**[*]**, Bashar NUSEIBEH**[**]**, Yasuyuki TAHARA**[++]**, Thein T. TUN**[**]**, Hironori WASHIZAKI**[^^]**, Nobukazu YOSHIOKA**[++]**, Yijun YU**[**]

[**] *The Open University, United Kingdom*
[++] *National Institute of Informatics, Japan*
[^^] *Waseda University, Japan*
[*] *University of East London, United Kingdom*
[+] *Shinshu University, Japan*

ABSTRACT

Addressing the challenges of developing secure software systems remains an active research area in software engineering. Current research efforts have resulted in the documentation of recurring security problems as security patterns. Security patterns provide encapsulated solutions to specific security problems and can be used to build secure systems by designers with little knowledge of security. Despite this benefit, there is lack of work that focus on evaluating the capabilities of security analysis approaches for their support in incorporating security analysis patterns. This chapter presents evaluation results of a study we conducted to examine the extent to which constructs provided by security requirements engineering approaches can support the use of security patterns as part of the analysis of security problems. To achieve this general objective, we used a specific security pattern and examined the challenges of representing this pattern in some security modeling approaches. We classify the security modeling approaches into two categories: problem and solution and illustrate their capabilities with a well-known security patterns and some practical security examples. Based on the specific security pattern we have used our evaluation results suggest that current approaches to security engineering are, to a large extent, capable of incorporating security analysis patterns.

1. INTRODUCTION

The collective experience of engineering secure software systems indicates that potential considerations for vulnerabilities in system design are both broad and deep. Anything from a single line of program code, the level of power consumption by the computer, to lapses in human memory may invite security breaches. Security engineers, therefore, need an array of tools at their disposal in dealing with diverse security problems. An integral part of the toolkit is the ability to access transferable design knowledge. Very often it is convenient to document this transferable knowledge in a pattern. A *pattern* is a description of a recurring problem and its corresponding successful solution [12]. As a pattern describes the identified recurring problem and its solution in principle, it (pattern) can be described in different languages. We call the description of a pattern using a specific modelling language, such as UML, its *representation*. Security patterns are well-understood solutions to recurring security problems [35]. They enable engineers to recognise, with relative ease, known vulnerabilities in their design and potential solutions. Several security patterns have been reported by practitioners and researchers, and there are lively and ongoing discussions about the discovery, documentation and application of security patterns.

Although many security patterns are documented in the public domain, they are often specifically tied to the language and the method in which they are expressed. Since security engineers do not have a common language and method to model, analyse and implement systems, it is important to know whether a particular security pattern can be expressed and applied in their own approach. This chapter aims to examine some of the languages in which security patterns may be expressed and the methods in which they are applied, with a view to articulating their relative strengths and weaknesses.

In this survey, we will focus primarily on the languages for modelling and methods for applying security patterns in early requirements analysis and designs. This choices are both principled and practical: principled because earlier patterns are less understood compared to those at the implementation level and because early

prevention of security vulnerabilities is thought to be less costly than remedial actions taken later; and practical because further expanding the scope of the survey would open up issues that are too many to be discussed in this chapter.

The main contribution of this chapter is an evaluation of security pattern modelling approaches. Our evaluation builds on the survey of security patterns by Yoshioka et al. [45]. We compare approaches from the following three categories because these approaches are repeatedly referred as representative ones that address security in models[48][49]: object-oriented design (UML, SecureUML, UMLsec, Misuse Cases), goal-oriented (KAOS, SecureTropos, i*), and problem-oriented (problem frames, abuse frames). In comparing and contrasting these different approaches, we adopt the widely acknowledge security pattern, Roll-Based Access Control (RBAC), and a familiar example to illustrate different aspects of each approach using a common set of evaluation criteria to judge the pros and cons of each approach. The general objective of this survey is to evaluate how security patterns can be described in selected requirements engineering approaches: in particular whether all key properties of RBAC can be expressed in those approaches. In other words, we are evaluating what each RE approach is able (and not able) to describe. Evaluation results should be useful to the following possible stakeholders using security patterns in their work:

Security pattern designers: who represent security patterns, can use the results to find out what languages are appropriate for modeling their patterns.

Application designers: who use security patterns to develop their applications with required quality characteristics including security, can use the results to improve understandings on existing security patterns that are modelled by one of the approaches evaluated in the chapter because the results provide what attributes of security patterns are described validly and how to interpret these representations.

The chapter is structured as follows. Section 2 defines the main characteristics of security patterns using the well-known dimensions of pattern languages, focusing on what makes a security pattern distinctive from general ones. In Section 3, we present a running example consisting of a security problem from the banking domain and RBAC as an example of a security pattern. Based on the security pattern evaluation criteria introduced in section 2, Section 4 reviews security modelling approaches and evaluate their capabilities to representing security patterns using the RBAC and the banking example introduced in section 3. Section 5 presents a comparative summary of the evaluation results from section 4. Finally, section 6 presents our conclusions, identifying open research issues in security pattern modelling and articulate an agenda for further research.

2. CHARACTERISING SECURITY PATTERNS

In software engineering a pattern documents an abstract relationship between a recurring software development problem that arises in a specific context and a well-proven schema for its solution [4]. Similarly, in security engineering a security pattern documents the description of a solution to a recurring security problem in a specific context [6]. In essence, security patterns are abstraction of real security problems and are identified through an analysis of real systems. In this section we present basic attributes of security patterns. By discussing these attributes we aim to address a question fundamental to the rest of the presentation of the rest of the chapter. This fundamental question is *what makes a security pattern*?

2.1 Characteristics of Security Patterns

According to Fernandez et al. [6], a security pattern has five main attributes, namely: the problem, context, forces, solution, and consequences. Besides these main attributes, others exist in a pattern template such as 'know uses' and 'related patterns'. In this survey chapter we will focus on the five main attributes as these are more relevant to our main objective. Worth noting from these main pattern attributes is the fact that each pattern is specific to a given type of generic problem, solution, and context. The rest of the section describes each of these attributes in detail and their rationale.

Problem: The problem part of a security pattern captures the *intention* of the patterns as a specific recurring problem that it is aimed at addressing; that is, what is it for? The problem at which the pattern is aimed can be classified by the type of security goals such as confidentiality, integrity, availability, or accountability. Depending on the type of security goal, security patterns capture different concerns; for example, the set of challenges relevant to a confidentiality problem are different from those of an availability problem. For this reason, a

security pattern must be specialised to certain types of security problems that it is intended to address. Being able to capture and analyse a security problem is an important characteristic of a security pattern modelling language.

Context: The characteristics of a security problem are influenced by its environment referred as its *context*. Context defines the conditions or situation under which a security problem may occur. It can be specified in terms of: the type of *attack* that could exploit vulnerabilities resulting from the identified security problem; the types of *assets* that could be affected (whether tangible or intangible) by an attack; and the type of *harm* or loss on assets that could occur as a result of a successful attack. Context is an essential characteristic in a security pattern as it enables the user of a pattern to evaluate its relevance and limitations to its application in a particular domain. Also worth noting is that a security problem may raise different concerns, depending on the context in which it considered. For example, addressing the integrity goal in a database of medical records may present different security issues from addressing integrity in a nuclear plant control system.

Forces: A security problem may have several potential solutions. Forces document the rationale for selecting one of these solutions. They state reasons for selecting a particular security solution from several potential solutions and capture the pros and cons (advantages and disadvantages) of applying the solution. Forces are critical for evaluating alternative solutions and for understanding the costs and benefits of applying the pattern. Such rationale helps the user of a pattern to appreciate why the particular solution was chosen to be part of the pattern.

Solution: A solution is a description of the machine (plan, task, action or structure) whose execution (or application to the context) can mitigate the security problem. The solution is well-proven in the sense that it has been validated to address the security problem adequately in previous context in which the problem was encountered. For example one solution to confidentiality of data transmitted over a public network is to use an encryption mechanism which reduces the risk of confidential information being disclosed to an attacker tapping on the communication medium. In the context of protecting data stored on a computer an authentication mechanism may be a more appropriate solution. The solution part does not need to describe a particular concrete solution to a security problem, because a pattern is like a template that can be applied in different situations. Instead, it describes how a general arrangement of elements solves the corresponding security problem in the pattern. This confidentiality example also demonstrates the role of context in selecting a suitable security solution to a security problem. Documenting the solution-part of a security pattern is important as it facilitates the reuse of solutions to commonly recurring problems. This serves the essence of having a security pattern.

Consequences: The application of a security pattern may result in changes in the context of application. Such changes are due to the fact that security requirements often conflicts with functional requirements and as a result the application of a security pattern may impact a system's flexibility, extensibility, portability, or usability. Consequences document the impact of the changes brought about by the application of pattern explicitly and help a security analyst understand and evaluate the capabilities of the given pattern.

2.2 Patterns of Problems versus Patterns of Solutions

In addition to the criterion described above, our evaluation of each security pattern modelling approach would also discuss the phase of application of the approach. In the context of software engineering approaches, security patterns can largely be classified into two categories, depending on the phase of software development where they can be applied. They can either be classified as patterns of security problems, patterns of security solutions, or both problem and solution. Patterns of security problems document recurring structures in analysis of software development problems in the problem space.

Examples of problem patterns are Jackson's problem frames [13] and their abuse frame extension [16, 17]. Once the security problem to be solved by the envisioned software system is well-understood, the next step is to move into the solution space to design the security solution. Designing and developing a security solution has its own set of problems which are determined by the characteristics of the chosen solution. The problem in the solution space should not be confused with the problem of understanding the security needs of stakeholders in the problem space. Patterns of security solutions document recurring structures of problems and their solutions in the solution space. Examples of solution patterns include architectural patterns such as pipe-and-filter [22] or design patterns in object-oriented software development [9].

We will use these characteristics of security patterns and evaluation criteria for comparing and contrasting approaches to security engineering. Our evaluation will look at the extent to which each approach is capable of supporting each of the attributes of security patterns discussed in section 2.1.

3. RUNNING EXAMPLE

This section presents a running example consisting of a security problem from the banking domain and RBAC as an example of a security pattern. We use this running example in the rest of the chapter for illustrating and comparing requirements languages that may be used in representing patterns.

3.1 Banking Problem Example

We use the "open account" use case in the bank example described by Fernandez et al. [8] to illustrate the features of the security analysis approaches surveyed. The execution of the use case involves the participation of two actors: a bank customer and bank manager. The example is about a bank customer opening a bank account securely.

The use case has the following main steps: (1) the customer provides personal information to the bank manager; (2) the bank manager performs credit checks; (3) if the customer has a good credit record, the bank manager creates an account; (4) the customer then makes an initial deposit into the account, (5) the bank manager creates authorisation and issues a card to the customer. Each of the steps in fulfilling the requirement of opening the bank account presents potential security vulnerabilities.

3.2 Role-Based Access Control

The selected security pattern example is Role-Based Access Control (RBAC). In RBAC, access control policy is embodied in user-role and role-permission relationships for achieving Separation of Duties (SoD). Users and permissions are not directly bound but indirectly associated via roles. When a user acts one of his/her roles under a session, he/she does not have any permission bounded with the other roles. For the sake of valuable comparison, RBAC pattern described in this paper includes the functionalities of the authorization.

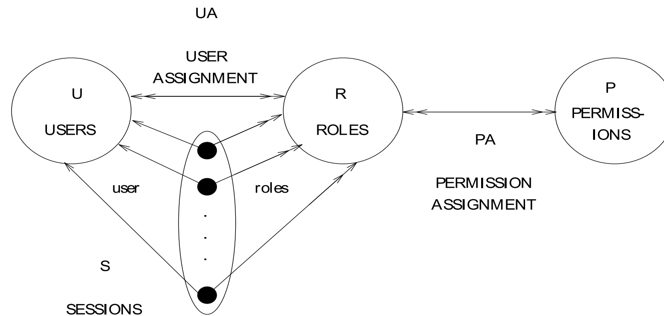


Figure 1: Structure of $RBAC_0$ (from [34])

For briefly understanding the structural aspect of RBAC, we explain $RBAC_0$ [34], - a simpler variant of the RBAC which is rich enough for our discussions. The structure of $RBAC_0$ is illustrated in Fig. 1. $RBAC_0$ is formalised as the four sets, two relationships, and two functions. The sets of entities U, R, P, and S denote users, roles, permissions, and sessions, respectively. The relationship between users and roles, called *user assignment*, is denoted by $UA \subseteq U \times R$. The *permission assignment*, $PA \subseteq P \times R$, also denotes the relationship between permissions and roles. Here, suppose that a session $s_i \in S$ is given; the associated users and roles are determined by the following functions: $user(s_i) \in U$ and $roles(s_i) \subseteq$. Consequently, the session s_i has all the permissions of $\{p \mid (p, r) \in PA\}$ for all $r \in roles(s_i)$.

RBAC is a well-understood and widely used security pattern. A unified model for RBAC is published as the National Institute of Standards and Technology (NIST) RBAC model [11] and adopted as an ANSI/INCITS standard [3]. RBAC is implemented in a variety of commercial systems, such as Sun's J2EE and Microsoft Windows.

4. SECURITY MODELLING APPROACHES

In this section we review security modelling approaches and evaluate their capabilities to representing security patterns using the RBAC and the banking example introduced in section 3 and the security pattern evaluation

criteria introduced in section 2. For each modelling approach we introduce only the aspects that are sufficient for the evaluation. As stated in the introduction, the main objective of this chapter is to survey requirements approaches to security patterns. We classify the approaches into three categories: design, goal-oriented, and problem-oriented. Under each category we selected several representative requirements engineering techniques such as problem frames, goals, and UML.

4.1 Design Approaches

Design approaches are based on the notion that models help requirements analysts in understanding complex software problems and identifying potential solutions through abstraction [10]. For example, models have been successfully for abstracting source code into class diagrams in reverse engineering. Such abstractions make it easier to understand the behaviour of a software system and how it might be improved. In this section we review four representative design approaches in security engineering, namely: UML, SecureUML, UMLSec, and Misuse Cases.

4.1.1 UML

In this subsection we summarize the UML approach, and show how it may be applied to illicit the security concerns.

(i) Overview of UML

Unified Modelling Language (UML) [32] is a widely used model notation method for mainly software and systems. UML defines several diagrams to express different aspects of software and systems in an abstract way, such as the class diagram for the static structural aspect and the sequence diagram for dynamic behavioural aspect.

Although UML does not originally cover non-functional characteristics including security in an explicit way, it is possible to analyse and represent vulnerabilities in the the target system and the vulnerabilities can be mitigated from the viewpoints of structure and dynamic behaviour.

(ii) Representing a Security Pattern in UML

In this subsection we show how the RBAC security pattern can be represented in UML. We also illustrate its application on the banking example. Our modelling of RBAC in UML is based on a meta-model proposed in Fernandez et al. [9, 35].

The RBAC pattern describes how to assign precise access rights to roles in an environment where control of access to computing resources is required such that confidentiality and availability requirements are preserved, and where there are a large number of users and a variety of resources. Rights of access to resources are assigned to roles instead of users directly through an authorization policy and users are assigned to roles. An object-oriented class structural model provided by the solution of RBAC is shown in Figure 4.1.1a. The User, Role, Right classes describe the users registered in the system, the predefined roles in the system, and the types of access rights to the protected computing resources (described by the ProtectionObject class), respectively.

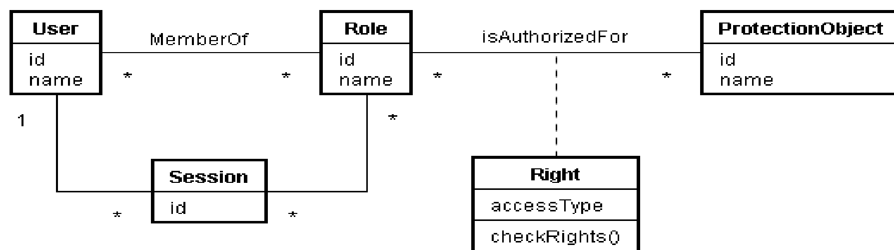


Figure 4.1.1a. Structure provided by the solution, in the form of the UML class diagram [35]

The user-to-role and right-to-role relations are many-to-many assignments. Among these relations, the separation of duties can be represented as an additional constraint on the user-to-role relationship. The Right

class has accessType (such as "read only") as its attribute, and a function for checking rights (i.e. permission) as its method (checkRights()). Usually the function receives an access request and returns the result of checking whether the request is permitted according to the accessType. The Session class describes temporal situations where the users activate a subset of the roles they belong to [34]. Each session must belong to a single user.

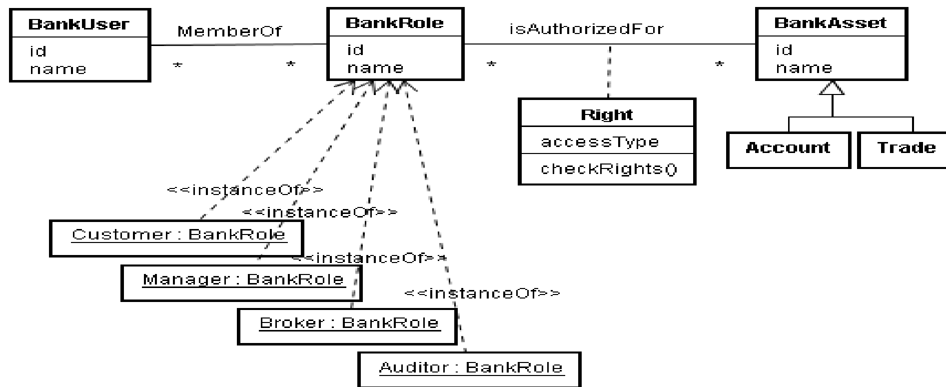


Figure 4.1.1b. Structure of the analysis model of the bank domain with RBAC

The application of RBAC to the bank domain is as shown Figure 4.1.1b. In this example, the main roles are the bank customer and the bank manager. Different rights are assigned to different actors (i.e. roles). For example, customers are granted to full access (including the transfer operation) to only their own accounts. Meanwhile, the bank manager is granted access to the customers' accounts in restricted ways.

(iii) Evaluation of Security Pattern Support in UML

In this subsection we evaluate the extent to which UML supports the representation of security concerns using the generic characterization of security patterns.

Problem: It is hard to model high-level security goals explicitly with UML. This is due to the fact that UML is a language for communicating designs rather than analyzing software problems.

Context in General: Although UML does not have any feature specific to security patterns, the structural and behavioural aspects of attacks and assets can be modelled by UML.

Context for Attack (Threat): It is possible to model concrete structures and behaviour of specific attack scenarios as threats with UML. This is achieved by using the dynamic behavioural constructs of UML such as sequence diagrams.

Context for Assets: It is possible to model dependencies among tangible and/or intangible assets and other entities from the viewpoint of structure or behaviour with UML. However it is hard to clarify the type of each asset.

Context for Harms to assets: It is hard to model characteristics and/or degree of certain harms to assets explicitly with UML.

Forces: It is hard to model reasons or rationales affected for the choice of the certain security solution from potentially several solutions with UML. Moreover, UML is not appropriate for modelling complex dependencies including alternative solutions.

Solution: It is possible to model structures and behaviour of the machine with UML. Moreover constraints (excluding temporal logics) on the machine and its environment can be represented by OCL included in UML 2.0 or later.

Consequences: It is hard to model tradeoffs and/or effects on quality characteristics including security explicitly with UML. The application of the RBAC pattern on a target environment makes it possible to control access to computing resources precisely while keeping high maintainability and low complexity because typically there are much more users than roles.

Table 4.1.1 presents a summary of the evaluation of the support for security patterns in UML based on the discussion above. UML tends to be used for modelling solutions mainly because it can capture structural and behavioural aspects of security functions and/or constraints on the environment, and sometimes referring to problems.

Table 4.1.1: Evaluation of Support for Security Patterns in UML (YES* denotes "Partially YES").

Problem	Context				Forces	Solution	Consequences
	General	Attack	Assets	Harms to Assets			
NO	YES*	YES	YES*	NO	NO	YES*	NO

UML provides a special diagram notation for representing parameterized collaboration, which can be used for modelling structure and behaviour of the solution of any pattern. Moreover, UML provides a built-in generic extension mechanism called UML Profile to customize UML models for particular domains by using additional stereotypes, tagged values and constraints for specific model elements. There are several UML profiles for patterns such as the UML Profile for Patterns as a part of the UML Profile for Enterprise Distributed Object Computing (EDOC) specification [31]. However, these existing diagram and profile are not specific to security patterns so that they are incapable of representing security concerns with precise semantics explicitly.

4.1.2 SecureUML

(i) Overview of SecureUML

Lodderstedt *et al.* [24] present a modelling language, based on UML, called SecureUML. SecureUML focuses on modelling access control policies and how these (policies) can be integrated into a model-driven software development process. It is based on an extended model of role-based access control (RBAC) and uses RBAC as a meta-model for specifying and enforcing security. RBAC lacks support for expressing access control conditions that refer to the state of a system, such as the state of a protected resource. In addressing this limitation, SecureUML introduces the concept of authorisation constraints. Authorisation constraints are preconditions for granting access to an operation.

The combination of the graphical capability of UML, access control properties of RBAC, and authorisation constraints makes it possible to base access decision on dynamically changing data such as time. Similar to its parent modelling language UML, SecureUML focuses on the design phase of software development.

(ii) Representing Security Patterns in SecureUML

The representation of security patterns in SecureUML inherits features from both UML and RBAC and is based on the following concepts: Role, Permission, ResourceSet, ModelElement, ActionType, and AuthorisationConstraints [24]. Figure 4.1.2a is a meta-model that illustrates how these concepts are related. The concepts of Role, Permission, ResourceRType, and ActionType are from RBAC and they are described in section 3. Meanwhile ModelElement is a UML concept.

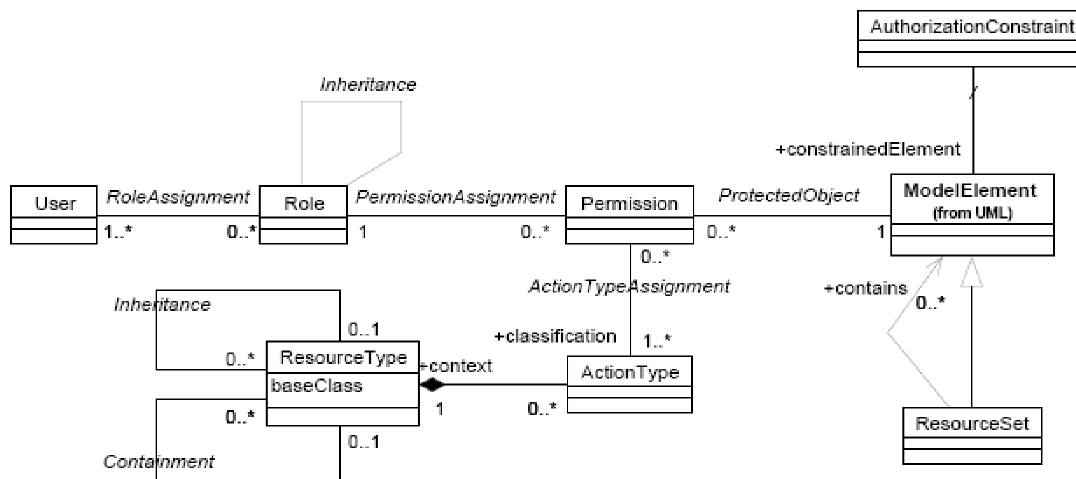


Figure 4.1.2a. SecureUML meta-model (from [24])

SecureUML introduced the concepts of ResourceSet and AuthorisationConstraint. A ResourceSet represents a user-defined set of model elements used to define permissions or authorisation constraints. An authorisation constraint is a part of an access control policy that expresses a precondition imposed on every call to an operation on a resource. AuthorisationConstraint is derived from the UML core type Constraint. The precondition depends on the dynamic state of the resource, the current operation call, or the environment.

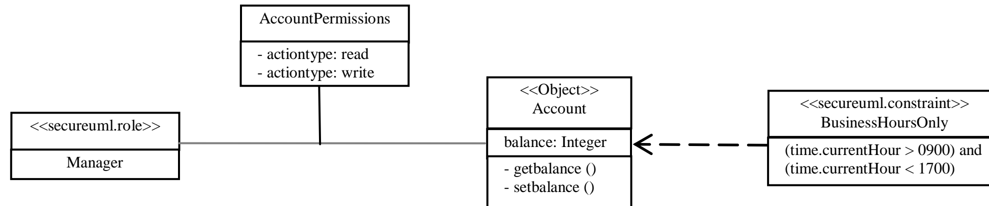


Figure 4.1.2b. Example of SecureUML constraints

Figure 4.1.2b illustrates an authorisation constraint which states that a manager can update and view a bank account, but such operations are limited to business hours only. As shown in the Figure, in SecureUML, authorization constraints are tagged with the stereotype <<secureuml.constraint>>. A role is identified with the stereotype <<secureuml.role>>. The Account class is an abstraction of the asset to be protected and it is identified with the <<object>> stereotype.

(iii) Evaluation of Patterns Support in SecureUML

Problem: SecureUML does not explicitly model security goals but focuses on modelling solutions to security problems. Its foundation of on RBAC implies that it is specific to security goals relating to controlling access to shared resources.
Context: The modelling of context in SecureUML is similar to RBAC. However, the context only captures assets that may be harmed in the event of an attack. It does not model scenarios of attacks and possible harm to assets.
Forces: There is no construct for capturing and modelling forces in SecureUML.
Solution: Yes. The combination of RBAC with UML and the authorization constraints extension is the bases of a security solution in SecureUML.
Consequences: Yes. The consequences of using SecureUML is a solution to an access control problem in access rights to resource are assigned to roles and users are assigned to roles with specific authorization constraints. The evaluation of security pattern support in this approach is summarized in Table 4.1.2.

Table 4.1.2: Evaluation of Support for Security Patterns in SecureUML

Problem	Context				Forces	Solution	Consequences
	General	Attack	Assets	Harms to Assets			
NO	YES	NO	YES	NO	NO	YES	YES

4.1.3 UMLsec

(i) Overview of the UMLsec

UMLSec [20] is an extension of UML which allows an application developer to embed security-related functionality into a system design and perform security analysis on a model of the system to verify that it satisfies particular security requirements. Security requirements are expressed as constraints on the behaviour of the system and the design of the system may be specified either in a UML specification or annotated in source code.

Automated theorem proving or model checking is used to establish whether security requirements hold in the design. If the design violates security requirements, a Prolog-based tool is used to generate a scenario (in the form of attack sequences) of how security requirements may be violated by the design and countermeasures are taken to remove the vulnerability. In essence, UMLsec assumes that requirements have already been elicited and there exists some system design to satisfy them. Its objective is to establish whether the system design

satisfies security properties. The design is then progressively refined to ensure that it satisfies security requirements.
.

(ii) Representing Security Patterns in UMLsec

UMLsec defines several new stereotypes towards formal security verification of elements such as: fair exchange to avoid cheating for any party in a 2-party transaction; secrecy/confidentiality of information; secure information flow to avoid partial leaking of sensitive information; and, secure communication links.

The UMLsec approach consists of two main steps. The first step is translating UML models into UMLsec specifications. UMLsec specifications describe the behaviour of a system in terms of its components and their interaction. The behaviour of system components is described in terms of the messages they exchange in communication links between them. The formal semantics of the communication between components are similar to Hoare's communication sequential processes (CSP) [16].

The next step, security analysis, involves eliciting ways by which an adversary may modify the contents of the data exchanged in communication link queues that may compromise the integrity of system behaviour. The analysis focuses on a consideration specific types of adversaries that may attack a system in a specific way. An example of such an attack on a communication link between components is breach of confidentiality, which state that some information will only become known only to legitimate parties. UMLsec specifications are checked for vulnerability to types of threats on contents of a communication link such as delete, read, and insert. The types of threats are adversary actions associated with particular adversary types. *Delete* means that an adversary may delete messages from a communication link queue. *Read* allows an adversary to read messages in the link queue, while *insert* allows the adversary to insert messages in the communication link.

The above discussion illustrates how security patterns are supported in UMLsec. In summary, first, UMLsec specifications are described based on component behaviour and patterns of interaction between system components. Secondly, the analysis for security vulnerabilities is guided by specific types of adversaries with specific classes of threats on contents of communication links. The classes of threats are also associated with specific types of security goals that an adversary may violate.

(iii) Evaluation of Patterns Support in UMLsec

Problem: Although security analysis is guided by specific goals and constraints in checking for security vulnerabilities in a system design, UMLsec does not have a specific construct for modelling security problems.

Context: Yes, the UMLsec approach explicitly models context of a security problem. However this context is limited to system design components, their interactions, and adversary models. Although this level of context consideration may be sufficient for identifying security vulnerabilities in system design, it is less useful for reasoning about vulnerabilities that arise due to threats initiated by the way the system is used such as an illegitimate transfer of funds by someone playing the legitimate role of a bank manager.

Forces: Once security vulnerabilities have been identified the system design is progressively refined to eliminate the threat. The rationale for selecting a particular solution of refining a design is not explicitly captured and it is not explicit whether alternative solutions are explored. It is possible though that such alternative security solutions can be explored in the refinement process based on the native UML design.

Solution: UMLsec provides an explicit refinement of design in order to ensure that they satisfy security constraints. Once a design has undergone refinement its ability to satisfy security requirements is re-verified. The refinement continue until it can be demonstrated that the vulnerability of the design to attacks is eliminated

Consequences: When a design has been found to violate security requirements, UMLsec provides for the generation of scenarios, in the form of attack sequences, which explain how security requirements may be violated by the design. The results (consequences) of refining a system design in order to address security vulnerabilities are captured in the revised version of the design and assessed against security requirements. A summary of evaluation results of UMLsec is presented in Table 4.1.3.

Table 4.1.3. Evaluation of Support for Security Patterns in UMLsec

Problem	Context				Forces	Solution	Consequences
	General	Attack	Assets	Harms to Assets			
NO	YES	YES	NO	YES	YES	YES	YES

4.1.4 Misuse Cases

(i) Overview of the Misuse Cases

Use cases document functional requirements of a system by exploring the scenarios in which the system may be used [19]. Scenarios are useful for eliciting and validating functional requirements [38, 43], but are less suited for determining security requirements – which describe behaviours not wanted in the system. Similar to anti-goals [39], misuse cases are a negative form of use cases and thus are use cases from the point of view of an actor hostile to the system [1, 2]. They are used for documenting and analysing scenarios in which a system may be attacked. Once the attack scenarios are identified, countermeasures are then taken to remove the possibility of a successful attack.

Although misuses cases are not entirely design-oriented as they represent aspects of both problems and solutions, they have become popular as a means of representing security concerns in system design. Worth noting is that they are limited by the fact that they are based only on scenarios. Completeness of requirements analysed through scenarios is not guaranteed as other scenarios by which the security of a system could be exploited may be left out.

(ii) Representing Security Patterns in Misuse Cases

Figure 4.1.4 shows some of the use cases and misuse case in the bank account example. Use cases are represented as clear ellipses while misuse cases are represented with the shaded ellipses. The <<threatens>> stereotype implies that the given misuse case is a threat to the satisfaction of the requirements of the corresponding use case. The notation we use for misuse cases is based on requirements engineering process proposed by Sindre and Opdahl [38].

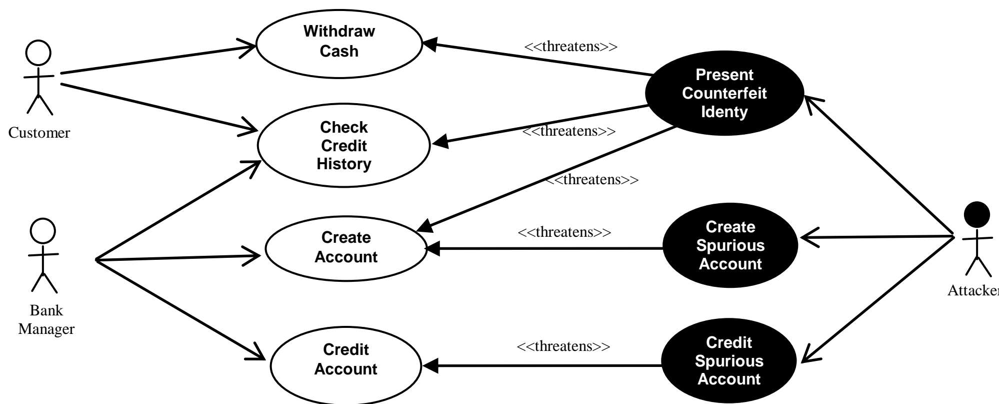


Figure 4.1.4. Use case and misuse cases in the banking example

As illustrated in Figure 4.1.4a, the security threats described in misuse cases are based on the functional requirements described in use cases. For example, the "create account" use case can be threatened by the "create spurious account" and "present counterfeit identity" misuse cases. The attacker in both misuse cases could be a malicious bank manager. An untrustworthy bank manager could also fraudulently transfer funds from a customer account to the spurious account.

(iii) Evaluation of Security Patterns Support in Misuse Cases

Problem: Misuses focus on describing security goals and refining them into corresponding security requirements. The identification of threats and definition of security goals in misuse cases tends to closely correspond to the use cases in which the threatened functional requirements are described. This bounds the identification of threats only to known functional requirements.

Context: Context is not explicitly captured in the high-level descriptions of misuse cases. It is possible though to describe assumptions about the states of the environment that make the misuse case possible. In the bank account example the "create spurious account" misuse case is possible if a user playing the role of a bank manager is untrustworthy. Similarly, the credit spurious account misuse case is based on the assumption that entities playing the role of a bank clerk may potentially divert funds from a genuine account to a fraudulent account. Hence, misuse cases describe assumptions about the problem context although these assumptions are

oversimplified and do not explicitly capture the context and the assets that can be harmed in the event of a successful attack [4].

Forces: Apart from the basic solution to satisfying a security goal, misuse cases have constructs for capturing alternative solutions called alternative paths. Alternative paths describe ways to harm the proposed system that are not accounted for by the basic path, but are similar enough that they can be described as variation to the basic path. Arguably, solutions to alternative paths are also potential solutions to the basic path. Although, alternative solutions can be explicitly captured through alternative paths there are no explicit means of comparing and contrasting their contribution to addressing the basic security problem with the exception of the fact that they are customized solutions for more specific scenarios. Hence there is no provision for capturing forces in misuse cases.

Solution: Misuse cases capture solutions for addressing security goals through *mitigation points* [38]. Mitigation points identify actions in a basic or alternative path where misuse threats can be mitigated and potential mitigation strategies that should be taken.

Consequences: The guaranteed outcome of mitigating a misuse case is described in a *mitigation guarantee* [38]. In this respect, a mitigation guarantee captures the consequences of applying a security patterns. The description in a mitigation guarantee depends on the level of detail of the description in mitigation points. If mitigation points are not specified in detail, then a mitigation guarantee describes the level of security required from the mitigating use cases to be described later. On the other hand if mitigation points have been described in detail, then a mitigation guarantee captures the strongest possible security guarantee that can be made.

Table 4.1.4. Evaluation of Support for Security Patterns in Misuse Cases

Problem	Context				Forces	Solution	Consequences
	General	Attack	Assets	Harms to Assets			
YES	YES	YES	NO	YES	NO	YES	YES

4.2 Goal-Oriented Requirements Approaches

4.2.1 Secure i*

(i) Overview

Distributed intentions, also known as the i* representations [46] are used to model and explore goal-oriented requirements of stakeholders in the problem space. Tropos is a process [6] that applies i* to analyse early requirements in order to come up with a validated list of specifications (tasks or plans) for a solution.

In i*, distributed stakeholders are categorised as agents, actors, roles and positions. Two kinds of relations among the goals of stakeholders are often analysed, one concerns the strategic rationale (SR) of individual stakeholders with AND-OR refinement respectively through decomposition and means-ends links; the other concerns the strategic dependencies (SD) among different stakeholders. Four types of intentions, namely goals, softgoals, tasks, resources, can appear in the SR model as nodes on the AND-OR refinement trees, or appear in the SD model as the dependum of the dependencies. Goals represent the desired states of the stakeholders, whilst soft-goals model quality requirements that do not have clear-cut Yes/No answers, such as security. The goals/tasks connect to softgoals through four types of contribution links (HELP + , HURT -, MAKE ++ or BREAK –) [46].

(ii) Security Patterns in i*

Figure 4.2.1a is an example i* diagram where intentions such as goals are shown as labelled ovals, tasks as hexagons, resources as rectangles, softgoals as clouds. The decomposition links are shown with arrowheads, whilst the means-ends links are shown with a mark of line segment, and the contribution links are shown with a type label. Agents are shown by dotted circles that enclose the intentions they contain. On the border of these circles are circular icons that indicate the type of the agent, such as Roles as shown in this figure. Strategic dependencies between two intentions of different agents are shown by arrows connecting them with an intention node beyond the boundaries of any agents. These dependency arrows have a D-shaped mark in the middle to emphasise their difference to decomposition/contribution links. Figure 4.2.1a is also an example i* diagram representing the RBAC security pattern [47].

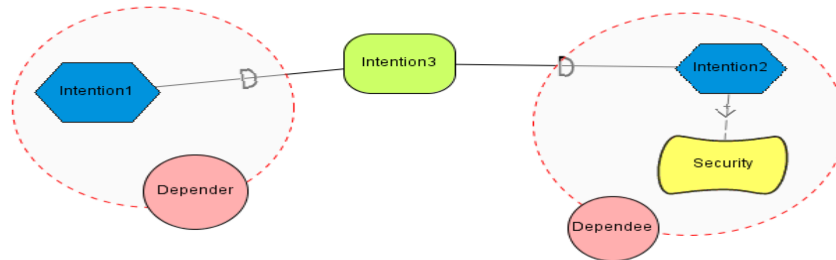


Figure 4.2.1a. A model showing the context of a security pattern in i*

One of the important issues in RBAC [4] is to clarify the responsibilities of two different roles. This can be represented by using two dependent roles "Depender" and "Dependee", and the "Dependee" contains a soft-goal "Security" in its SR model (Figure 4.2.1a). The achievement of "Intention2" of "Dependee" depends on the achievement of "Intention1" through a dependum "Intention3". If the role "Depender" may achieve "Intention1" by itself, there is no need for the role "Dependee" to achieve "Intention1". According to the RBAC security requirement, however, only the role "Dependee" is allowed to achieve "Intention1" to separate the duty. To avoid a violation of the separation of duty requirement in RBAC, one therefore must prevent "Intention1" from being executed by the role "Depender".

Hence, we introduce the following security pattern in i* by adding a negative contribution (HURT or BREAK) from "Intention1" to "Intention2", as introduced in Figure 4.2.1b. Since one cannot have "Intention2" both satisfied and denied when "Intention1" is satisfied, it avoids the vulnerability in the model. This is the consequence of the security pattern in i*.

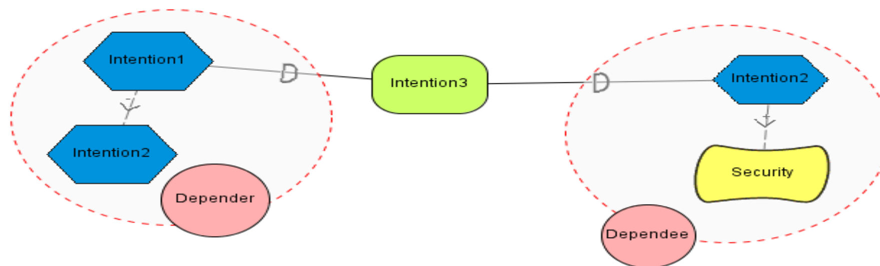


Figure 4.2.1b. The security pattern in i* to enforce responsibility

Figure 4.2.1c illustrates a realistic context for opening a bank account where the security pattern in Figures 4.2.1a and 4.2.1b can be applied. A task "manage PIN" may be achieved by a "Bank Clerk", but it seems to be insecure because "Bank Clerk" can maliciously manage the PIN by, e.g., issuing a very simple PIN or leaking the PIN. Consequently, the task "mange Personal Info" cannot be achieved by "Bank Clerk" because otherwise the personal information could have been abused to open the account by the malicious bank manager.

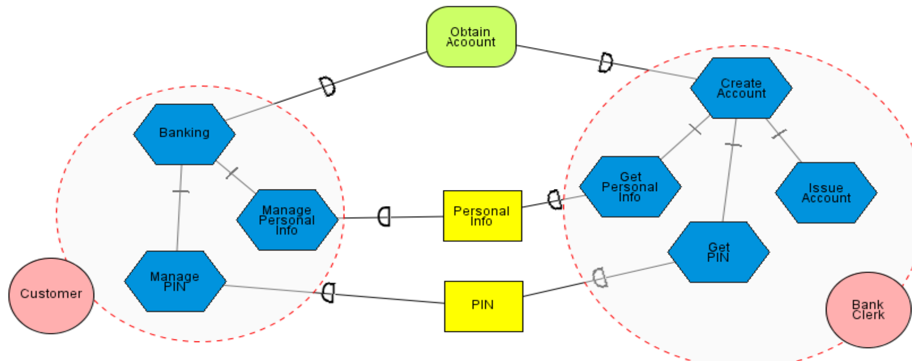


Figure 4.2.1c. A part of a Model for Opening Bank Account

To prevent this from happening, we apply the security pattern that results in a new securer model in Figure 4.2.1d, which satisfies the security property by the RBAC requirement.

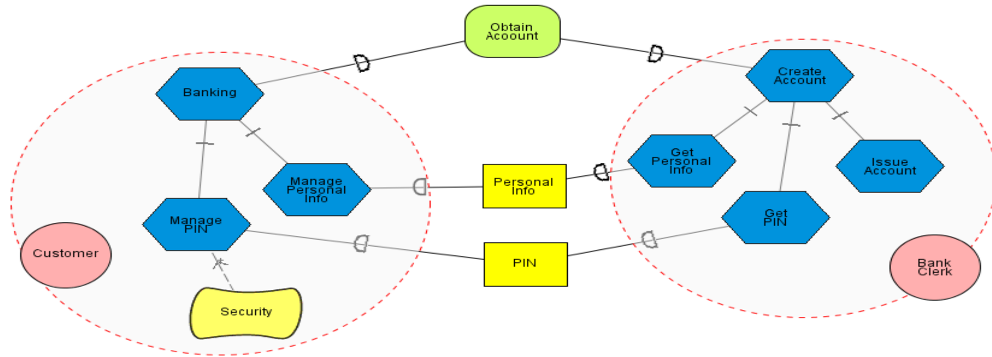


Figure 4.2.1d A new secure model

(iii) Evaluation of Patterns Support in i*

This subsection presents an evaluation of the extent to which security pattern modelling is supported in the i* approach. Table 4.2.1 presents a summary of the evaluation results based on the examples representing security patterns presented in the subsection 4.2.1(ii).

Table 4.2.1. The strength of support of security pattern using the i* approach

Problem	Context				Forces	Solution	Consequences
	General	Attack	Assets	Harms to Assets			
YES	NO	NO	NO	NO	YES	NO	YES

Problem: Supported. The security goals are represented natively as softgoals in the i* approach in addition to the dependencies among them. This is a strength in the i* approach for security patterns. The vulnerability in the context is specified in the i* diagrams through goal-based reasoning analysis, which is already part of the validation process in the i*/Tropos methodology. For example, whether the "security" softgoal is satisfied or denied can be evaluated by a bounded SAT-solver [36], and which tasks prevent the softgoal from being satisficed can be diagnosed based on the algorithms in [42]. The reasoning-based requirements analysis is one of the strengths in goal-oriented approaches.

Context: The i* diagrams (e.g., Figures 4.2.1a and 4.2.1c) present the situation when the security pattern of an i* approach is to be applied. Figure 4.2.1a is the general template of the security pattern whilst Figure 4.2.1c is an instantiated model where the intention names are instantiated by concrete names (e.g., Intention1="account opened"). Although it is possible to identify such a relationship between the template parameters intentions and the instantiated intentions from the context i* diagrams, it is however not represented explicitly in visual form. One needs to provide such mapping pairs in a separate list. The security-specific contexts, such as attacks, assets and harms, are not explicitly classified by the i* meta-model. These notations are considered as domain knowledge in i*.

Forces: The softgoals such as "Security" must be made explicit in the i* based security patterns. If not, there is one approach based on anti-goal analysis proposed by Liu et al [23] to elicit possible counter-models based on a "normal" goal model. Other non-security forces that may be used in trade-offs analysis are also represented by softgoals, making it suitable to use i* as the pivotal representation for trade-offs analysis. For example, the NFR framework has classified Security further into Availability, Confidentiality, Authentication, etc. Such decomposition patterns are a strength of i* approach. One limitation of such analysis in the qualitative reasoning based on the i* meta-model can be improved by its extension, Secure Tropos, using quantitative evaluations for risk management [13].

Solution: A plan for satisfying the security softgoal can be obtained by the application of reasoning algorithms on the i* diagrams. An exhaustive enumeration of all possible solutions can be found by several SAT-solver based algorithms. On the other hand, the exponential number of possible solutions may require a sophisticated trade-off analysis with respect to explicit contribution links to various forces (softgoals).

Consequences: Deriving the consequence contexts from i* is not natively supported for general patterns. However, it is possible with the model-driven transformations for well-defined security patterns such as RBAC [47].

4.2.2 Secure Tropos

(i) Overview of Secure Tropos

Secure Tropos [26-29] extends both the i* modelling language and its Tropos development process [5] . The main concept introduced by the secure Tropos is that of *security constraints* [27], which directly represent security requirements. The Secure Tropos methodology also provides a number of modelling activities for developers to analyse, delegate and decompose security constraints. Security constraints are satisfied by secure entities, which describe any goals and tasks that are related to the security of the system. Representing the strategic interests of an actor in security, *secure goals* model the high level goals an actor employs to satisfy any imposed security constraints. A secure goal can be achieved by an actor in various ways since alternatives can be considered [27].

The precise definition of how the secure goal can be achieved is given by a *secure task*. *Secure dependency* introduces security constraint(s) that must be respected by actors for the dependency to be satisfied. Both the depender and the dependee must agree for the fulfilment of the security constraint(s) in order for the secure dependency to be valid. This means the depender expects the dependee to satisfy the security constraint(s) and also that the dependee will make an effort to deliver the dependum by satisfying the security constraint(s). A graphical representation of a Secure Tropos model with the above concepts is shown in Figure 4.2.2a.

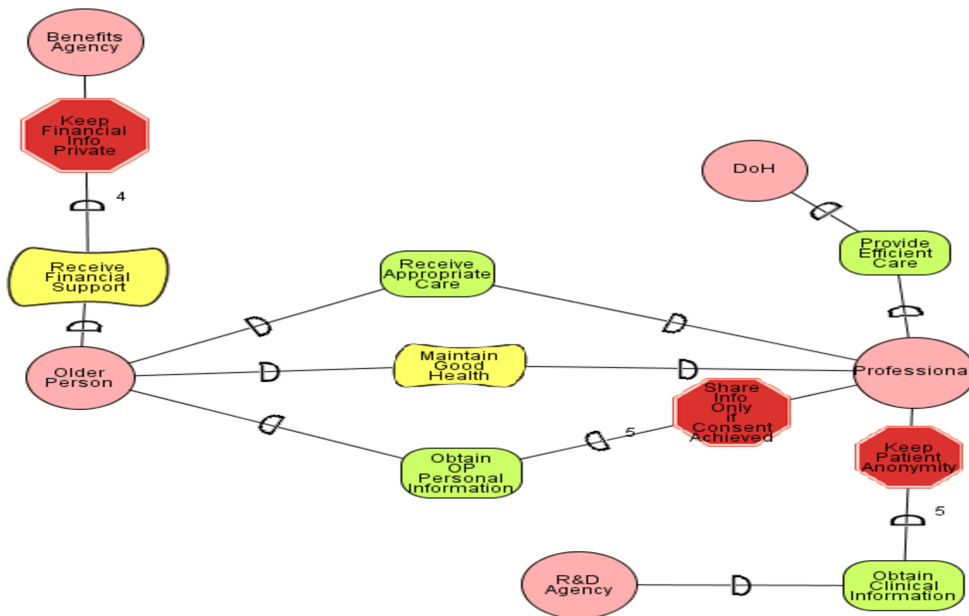


Figure 4.2.2a: A Secure Tropos model.

The process of Secure Tropos analyses the security needs of the stakeholders and the system in terms of secure dependencies and security constraints imposed to the stakeholders and the system; of identifying secure entities that guarantee the satisfaction of the security constraints; and of assigning capabilities to the system towards the satisfaction of the secure entities [27]. This process is spread out into four main phases: early requirements analysis, late requirements analysis, architectural design, and detailed design. During the *early requirements analysis* phase the secure dependencies between the various stakeholders of the system are analysed. In particular, secure dependencies and security constraints are identified for the various actors. During this stage, imposed security constraints are expressed, initially as high-level statements which are later further

analysed. Then secure goals and entities are introduced to the corresponding actors to satisfy the security constraints.

During the *late requirements analysis* phase, security constraints are imposed on the system under development. These constraints are further analysed according to the analysis techniques of secure Tropos and secure goals and entities necessary for the system to guarantee the security constraints are identified. During the *architectural design* any possible security constraints and secure entities that new actors might introduce are analysed. Additionally, the architectural style of the information system is defined with respect to the system's security requirements and the requirements are transformed into a design with the aid of security patterns. Furthermore, the agents of the system are identified along with their secure capabilities. During the *detailed design* phase, the components identified in the previous development stages are designed. In particular, actor/agent capabilities and interactions are specified taking into account the security analysis that took place in the previous stages.

(ii) Representing Security Patterns

In Secure Tropos, patterns are normally used to assist developers to identify a network of actors or a set of agents to solve specific security problems based on the security requirements of the system. In doing so, patterns are documented using a template that it is mostly based on the Alexandrian format. Each pattern is described in terms of its problem (intent), context, forces, solution and consequences. Depending on the targeted solution paradigm (network of actors or set of agents) developers can describe the patterns using standard or agent-oriented terminology and concepts (e.g. agency: the place where an agent resides). The solution proposed by any pattern in Secure Tropos is described in terms of social dependencies and intentional elements. This makes it possible to achieve a good understanding of the pattern's social and intentional dimensions - two factors very important for security.

In the context of the *Bank Example*, the agency represents a *Bank* and agents represent different individuals playing different roles such as *Customer*, *Bank Cashier*, *Bank Manager*. Assuming that an account is open, a customer might want to access the account information (balance, transactions, etc.) In doing so, the customer depends on the Bank. This simple dependency is represented as shown in Figure 4.2.2b. Following security related analysis in Secure Tropos, a number of security constraints are identified. For simplicity, in this example we keep the number of security constraints to a minimum. In particular, the customer is given the security constraint "to Keep Account Information Secure" and the Bank is required to maintain two security constraints "to Keep Customer Information Private" and "to Allow Account Access Only to Authorised Customers" as shown in Figure 4.2.2c. Once security constraints have been identified, it is important that a design is developed to fulfil such security constraints. It is at this stage that security patterns are applied.

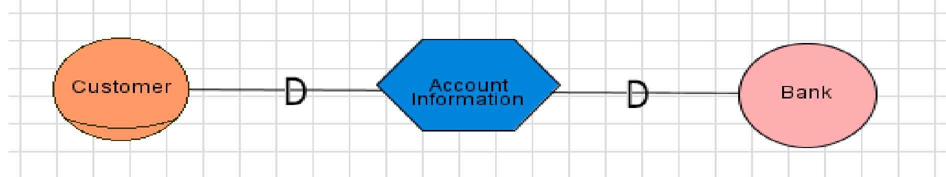


Figure 4.2.2b: A Simple Dependency for the Bank Example

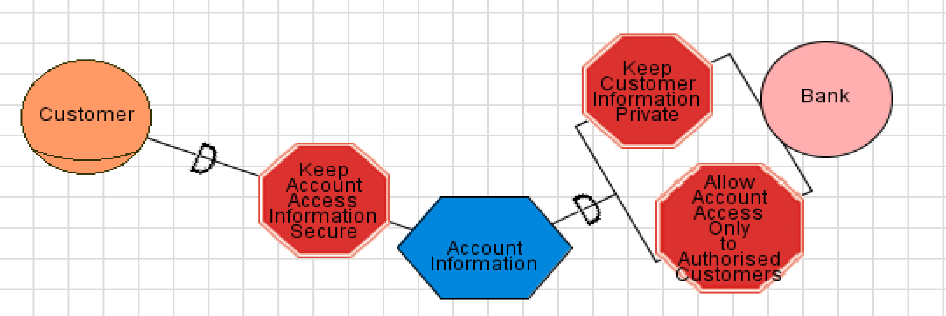


Figure 4.2.2c: A Secure Dependency for the Bank Example

In particular, the RBAC pattern can be used to create a design that satisfies the "Allow Account Access Only to Authorised Customers". The rest of the subsection illustrates how the RBAC pattern is represented using the Secure Tropos approach and how the pattern assist in developing a design that fulfils the appropriate

security constraint. It is worth mentioning that Secure Tropos adopts agent-oriented terminology and concepts when describing the pattern. This enables us to demonstrate how the approach deals with agent related security patterns and it also provides a slightly different view of the pattern as described in the rest of the approaches. This, in turn, allows us to demonstrate a wider application domain for the Secure Tropos approach and its representation of patterns, and enables us to evaluate the approaches described in this chapter in a wider context.

Problem: Allow an agency to provide access to its resources, according to its security policy, based on the roles that each agent plays. Agents belonging to an agency might try to access resources that are not allowed. Allowing this to happen might lead to serious problems such as disclosure of private information or alteration of sensitive data. How can the agency make sure that agents only access resources which they are allowed to access?

Context: Many agents exist in an agency. These agents often play different roles and most likely will require access (according to the role they play) to some of the agency's resources in order to achieve their operational goals.

Forces: It is unlikely that the access control facilities of all internal resources are activated and configured appropriately. In particular, out-of-the box installations offer standard services that can be misused by malicious agents. Even if there are restrictions to access, it is unlikely that they are consistent, especially when more than one administrator is involved and there are no "global" guidelines. Even worse, it could be assumed that most internal resources are not hardened. Experience shows that patches are not applied in time and that many, often unneeded services are running. Furthermore, it might happen that attacks cannot even be detected, as one cannot ensure that the audit facilities of the internal resources are activated and configured appropriately.

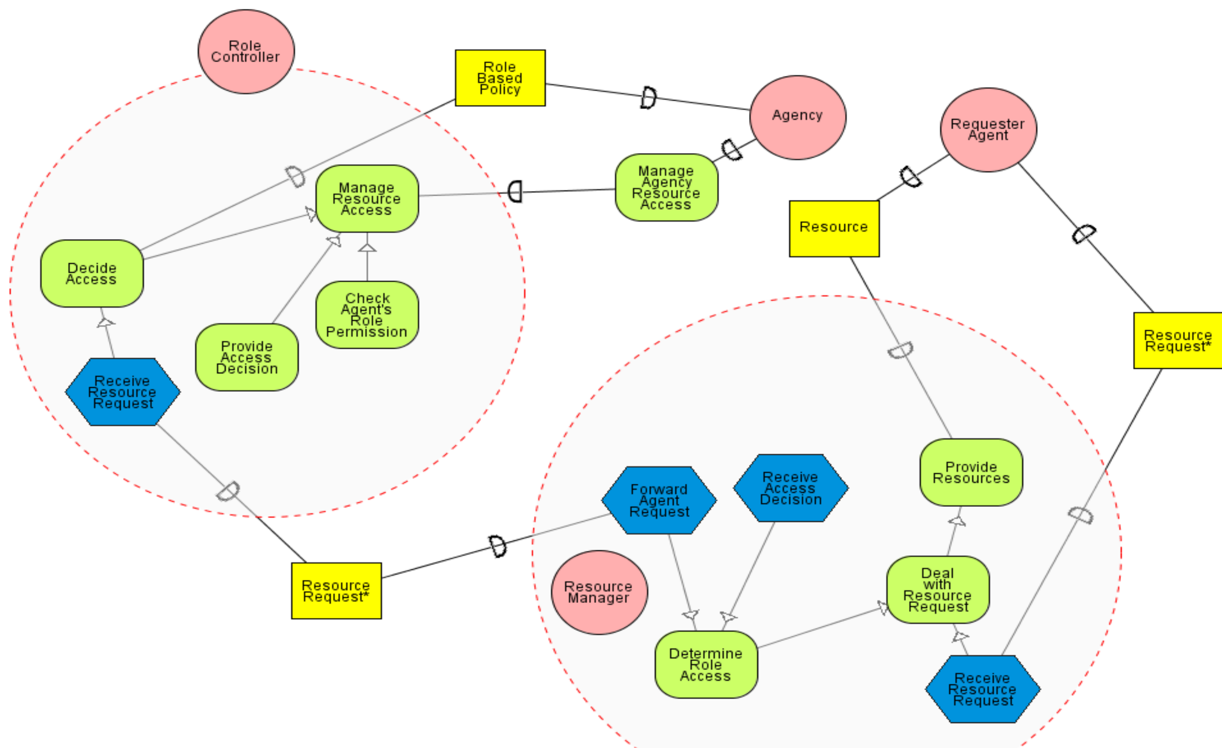


Figure 4.2.2d: Pattern Solution representation

Solution: Each agent is allowed access to a resource according to the role they play. A Role Controller agent exists in the Agency to enforce the role based policy of the Agency. Thus, when an agent, playing a specific role, requests access to a resource; this request is forwarded to the Role Controller agent. The Role Controller checks the role based policy against the agent's role and determines whether the access request should be approved or rejected. If the access request is approved the Role Controller forwards the request to the Resource Manager. The graphical representation of the pattern dependencies is shown in Figure 4.2.2d. The Requester Agent depends on the Resource Manager for the resource, and the Agency depends on the Role

Controller for checking the request. The Role Controller depends on the Agency for receiving the role based policy and for forwarding the request, which is forwarded to the Resource Manager in case it is approved.

Consequences: Agency's resources are accessed only by agents which playing specific roles. Different role based policies can be used for accessing different resources. However, a possible attack is that, if this fails the role base access control system fails.

4.2.2.3 Evaluation of Security Patterns Support in Secure Tropos

As mentioned before, security patterns can assist developers to develop a design that satisfies specific security constraints. Therefore, the use of security patterns in Secure Tropos is particularly encouraged during the architectural design stage of the methodology. Although an extended Alexandrian format is used to represent the security patterns in Secure Tropos, the real novelty of the pattern representation template used above is based on the presentation of the solution in terms of social dependencies of the actors. Such representation enables one to directly fit security patterns to Secure Tropos models. Going back to the *Bank Example*, the *Requester Agent* of the pattern solution representation is effectively the *Customer*, the *Agency* represents the *Bank* and the *Resource Manager* and *Role Controller* represent roles within the bank that are required for the RBAC to work such as *Account Manager* and *Bank Cashier* (or an electronic system in case request is made online or through a system).

It is now well recognized that security is not just a technical issue but it also demonstrates a social dimension. Therefore, to completely represent security patterns, it is important that Secure Tropos model the social dependencies that a security pattern introduces as part of its solution. Moreover, since the security aware concepts of the methodology contribute to the various security related analysis (attacks, assets, harm etc), the Secure Tropos allows one to specify contexts or applicability of the pattern and to quantify the risks. It is also important to mention that the methodology supports not only the representation of individual security patterns, but the representation of security pattern languages and the formalization of the properties of the patterns that belong to the language as demonstrated [30]. A number of guidelines on the application of patterns have also been defined [30]. Table 4.2.2.1 summarises the support provided by the Secure Tropos approach to the representation of security patterns.

Table 4.2.2: Summary of Evaluation of Pattern Support in the Secure Tropos approach

Problem	Context				Forces	Solution	Consequences
	General	Attack	Assets	Harms to Assets			
YES	YES	YES	YES	YES	YES	NO	YES

Secure Tropos is an improvement over i* on the Forces and Context attributes. Forces in Secure Tropos fully consider the quantified risks. It also improves partially the Context attribute of security constraints by providing additional contexts for security patterns.

4.2.3 KAOS

(i) Overview of KAOS

Introduced by van Lamsweerde at al., the KAOS is a requirements engineering framework that supports patterns of goal refinement that allow high-level goals to be stated in terms of a combination of lower level ones [7, 40, 41]. A goal is defined as statements that express the intended behaviour of the system under development, and in general it is expected that this behaviour will be achieved through the cooperative interaction of the agents that make up the system. Agents are the active components of the system, be they humans, devices, existing software or software-to-be, that will play some role in satisfying the goals of the system.

The goals can be specified in KAOS using both a formal and informal notation. The informal definition is specified is natural language whilst the formal definition uses the temporal logic notation introduced by [25]. KAOS provides reusable patterns of goal refinement which are formally proven in terms of temporal logic expressions. The approach taken for proving a given pattern is to assume that each of the sub-goals holds and then show that it is possible to infer the truth of the base goal from the conjunction (or disjunction) of the sub-goals. The patterns ensure that each stage of the elaboration process is correct, i.e. achieving the low level goals is equivalent to achieving the higher-level one; consistent, meaning that it is possible to satisfy all the low-level goals; and minimal, i.e. there are no redundant goals in the refined set.

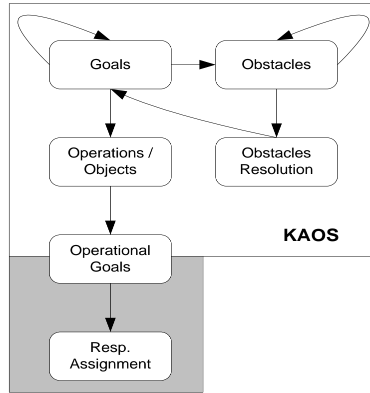


Figure 4.2.3a: KAOS goal elaboration process

KAOS represents each goal as a Temporal Logic rule and then makes use of refinement patterns to decompose these goals into a set of sub-goals that logically entail the original goal (Figure 4.2.3a). Obstacles were introduced into the KAOS framework in order to help identify scenarios that might cause goal violations. Obstacles are essentially negated goals that are used to prompt thinking about requirements in terms of states that the system should not achieve. Like goals, obstacles will at first be defined at a high-level and need to be elaborated into more precise definitions. The KAOS methodology provides techniques for resolving these lower-level obstacles by introducing new goals that ensure that the obstacles are avoided.

Once a goal has been elaborated to the level of a system-level requirement (i.e. an operational goal), the final stage of the procedure is to assign each of the refined goals to a specific object/operation such that the final system will meet the original requirements. KAOS defines a library of domain-independent refinement patterns, backed up by logical proofs that can be used to refine goals and obstacles (Figure 4.2.3b).

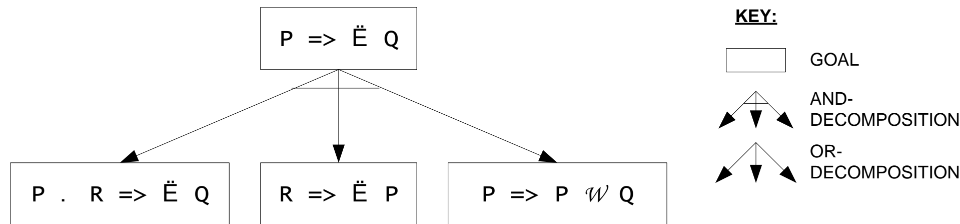


Figure 4.2.3b: Example of KAOS goal elaboration pattern

Once a refinement pattern has been derived, it can be applied to any matching scenario without the need to recreate the proof again. When applying a goal elaboration pattern, the system will present the user with the set of patterns that are valid for the given higher-level goal. Then it is up to the user to select a pattern that can be instantiated with meaningful values for each of the missing goal properties in the sub-goal formulae.

(ii) Representing a Security Pattern in KAOS

The KAOS approach provides support for security goal specification in terms of a number of specialised meta-classes of goal, namely, *Confidentiality, Integrity, Availability, Privacy, Authentication* and *Non-repudiation* goal subclasses. In order to support the concepts of attacker knowledge, the formal language of goals is extended with the epistemic operators, KnowsVag, which is defined as follows:

$KnowsV_{ag}(v) = \exists x: Knows_{ag}(x=v)$ ("knows value")

$Knows_{ag}(P) = Belief_{ag}(P) \wedge P$ ("knows property)

The operational semantics of the epistemic operator $Belief_{ag}(P)$ is defined as "P being one of the properties stored in the local memory of agent ag". The knowledge of a value of a property at a given point depends on

both the agent having a value for the property in its local memory *and* that property value actually holding at the given point in time.

The use of obstacles for security goals makes obstacle refinement trees analogous to the threat trees that are used for modelling potential attacks security-critical systems. However, obstacles neither capture the goals and knowledge of a potential attacker; or the vulnerabilities in software systems. The notions of anti-goals and anti-models were introduced to the KAOS framework in order to deal with these problems [40]. Combining with the epistemic operators described above, allows security patterns to be expressed in the KAOS framework. We illustrate this using the RBAC example presented previously.

In the banking example, the requirement is to ensure that only authorized users are allowed to perform particular operations on bank assets, such as accounts. We can document this requirement as the following top-level goal

Goal *Maintain*	[OperationOfAccountOnlyByAuthorizedUser]
InformalSpec	If user performs an operation on an account, they must be authorised to do so
FormalSpec	∀ acc:Account, op:Operation, u: User
	Supports(acc, op) ∧ Authorized(u, op, acc) => CanDo(u, op, acc)

A security pattern that documents a solution to satisfying this high-level goal, must take into account the potential attacks on the system that would negate the above, as shown in the following anti-goal.

AntiGoal *Achieve*	[UnauthorizedUserOperatesAccount]
InformalSpec	A user performs an operation on an account, with no authority to do so
FormalSpec	∃ acc:Account, op:Operation, u:User
	Supports(acc, op) ∧ Unauthorized(u, op, acc) => CanDo(u, op, acc)

This top level anti-goal identifies the asset and the capabilities of the attacker, and provides the general context for the security requirement being modelled. By decomposing the anti-goal, we can elaborate particular threats that would lead to the unauthorised operation of an account. The maintainability problem of the direct user-permission assignment model is highlighted by the following anti-goal:

AntiGoal *Achieve*	[FormerEmloyeeAuthorizationUnrevoked]
InformalSpec	A user retains permission to perform operation even after leaving employment
FormalSpec	∃ acc:Account, op:Operation, u:User
	[ExEmployee(u) => Unauthorized(u, op, acc)] ∧
	[Supports(acc, op) ∧ Unauthorized(u, op, acc) => CanDo(u, op, acc)]

RBAC provides a countermeasure to the above anti-goal by decoupling users from permissions using roles. Thus, permission revocation can be achieved by simply removing a user from a particular role. We can document the RBAC goal as follows:

Goal *Maintain*	[RBACModelForBank]
InformalSpec	Maintain a role-based access control model for the bank
FormalSpec	∀ acc:Account, op:Operation, u: User, ∃ r: Role
	Supports(acc, op) ∧ Member(u, r) ∧
	Authorized(r, op, acc) => CanDo(u, op, acc)

Sub-goals of this can be used to instantiate specific roles, operations and authorizations. A particular advantage of the KAOS approach to security requirements patterns is that the formal specification can be used to formally prove the correctness of the goal refinement. This allows the final pattern to be reused with confidence.

(iii) Evaluation of Security Pattern Support in KAOS

The KAOS framework allows security requirements patterns to be expressed in terms of the goals of the attacker (anti-goals) and vulnerabilities of the system under study. Patterns can also include a definition of the solution, or counter-measure, to the attack in terms of goals that avoid a given vulnerability. In this subsection

we evaluate the extent to which KAOS the representation of security patterns using the generic characterization presented in section 2.

Problem: The intent of a security requirements pattern expressed in KAOS is documented in the top-level goal of the pattern. The meta-class of the top-level goal will identify if the pattern pertains to a confidentiality, integrity, availability, privacy, non repudiation or authentication concern. The anti-goal model that forms part of the pattern definition can be used to identify the problem addressed by the pattern. In the example given above, the low-level anti-goal relating to permission revocation presents the permission maintainability problem that is addressed by the RBAC pattern.

Context: As with the intent, the general context of the problem the pattern aims to address will be documented in the top-level anti-goal. More specific details of the attacker knowledge, intention and asset properties will be captured in lower level goals of the pattern definition. The notation doe not provide an explicit means of specifying harms to assets, although these can be captured as annotations to the anti-goal model.

Forces: The KAOS pattern notation does not provide an explicit means of capturing the forces that might influence the selection of a particular refinement pattern. However, requirements engineers are able to use the preconditions specified in the formal definition of goals to determine the suitability of a give pattern for the problem at hand.

Solution: The KAOS pattern notation allows specification of the solution to the initial problem in the form of sub-goals that satisfy the original goal. In the RBAC example, the solution is denoted by the RBACModelForBank sub-goal, which can be further refined into specific instances of roles, operations and authorisations for a given scenario.

Consequences: The consequence of a KAOS refinement pattern is to satisfy the original, high-level goal. If a pattern is specified using the formal notation provided by KAOS, the entailment relation between the sub-goals and top-level goal can be formally proven. This ability to validate that the consequences specified for a given pattern are correct is particularly useful in the domain of security patterns.

Table 4.2.3: Evaluation of Support for Security Patterns in KAOS

Problem	Context				Forces	Solution	Consequences
	General	Attack	Assets	Harms to Assets			
YES	YES	YES	YES	Not Explicitly	Not Explicitly	YES	YES

4.3 Problem-Oriented Approaches

Problem oriented approaches [14, 15, 18], bring informal and formal aspects of software development together in a single theoretical framework for software engineering design – presenting software development as the representation and step-wise transformation of software problems. This theoretical framework allows for: identification and clarification of system requirements; the understanding and structuring of the problem world; the structuring and specification of a machine that can ensure satisfaction of the requirements in the problem world; and the construction of adequacy arguments, to convince both developers and other stakeholders that the system will provide what is needed. In this section we review two problem-oriented approaches, namely: problem frames [17] and abuse frames [21, 22].

4.3.1 Problem Frames

In this subsection we summarize the Problem Frames approach, and show how it may be applied to illicit the privacy and security concerns.

(i) Overview of Problem Frames

Introduced by Jackson [17], the Problem Frames approach (PF) provides a intellectual structure for analyzing software problems in the problem space. There are a few principles that are relevant this discussion. First, PF emphasizes the need for separating three descriptions: *specification*, *problem world domains*, and the *requirement*. Roughly, a specification (S) is a description of a software system that, within the context of certain problem world domains (W), satisfies a requirement (R). A problem diagram describing the relationship between W, S and R are typically described as shown in Fig. 4.3.1a. P_s is the phenomena shared between the machine and problem world, while P_r is the requirements phenomena. Second, in PF, a complex problem is decomposed by fitting its subproblems into of known problem patterns called problem frames. A frame captures the contextual

structure of a problem and concerns associated with the frame. One of such concern is the `proof obligation' to show W, S ⊨ R.

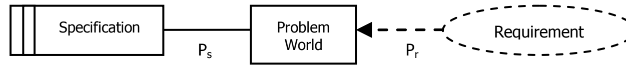


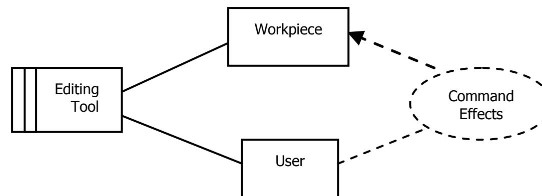
Figure 4.3.1a Separation of Descriptions in Problem Frames

The philosophy of problem frames is that some software development problems are recurring. Based on this premise, the main idea of problem frames is to document classes of commonly recurring problem structures and their solutions in problem-solution patterns. When a problem that matches a well known problem structure is encountered, the solution part of the pattern can then be re-used to solve the problem at hand.

(ii) Representing a Security Pattern in Problem Frames

The way patterns or frames are represented in PF has the following characteristic. Each frame is a generalization of some instances of recurring software problems. Frames tend to focus on the structure of problems rather than the solutions, although some work has been done to explore how the problem and solution structures are related. In each frame, there are three descriptions: the requirement, the problem world domains and the machine. A potential security vulnerability may be identified as a "concern", which may be attached to either the frame, the machine, the problem world domains and their connections, or the requirements itself. It means that vulnerability may arise from any of the elements.

One of the several problem frames currently recognized is called *workpiece frame*. Fig. 4.3.1b shows the structure of software certain problems in which an operator needs to use a computer program to edit a lexical object such as a text document, a picture, a variable and so on. The main concern of the problem is to take in certain commands from the operator, interpret them appropriately, and if the commands are valid, operate on the lexical object accordingly.



Concerns	**Descriptions**
Workpiece Frame Concern	Desired effect of user actions on the workpiece
Disobedience Concern	The user not following the appropriate course of actions
……	……

Figure 4.3.1b Workpiece Frame and Some Known Concerns

In PF, the account opening problem (without the security concerns first) fits a frame called the Workpiece frame. Fig. 4.3.1c shows how the problem of "opening of a bank account" may be represented in PF.

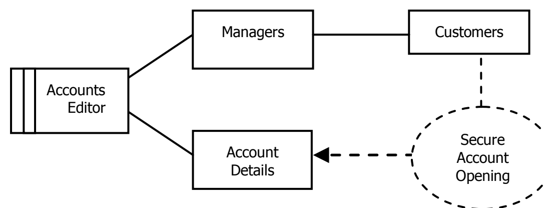


Figure 4.3.1c Problem Diagram for "Opening a Bank Account"

Informal descriptions of W, S and R are as follows:

R “opening of a bank account": if the customer is new, create a new bank account with the customer's details.

W *Customers*: people who provide their details to open a new bank account.
Managers: people who use the software to open a new back account.
Account Details: a storage where the details of the customer who opened a bank account is kept.

S The software examines whether the system already has opened an account for the customer, and if not, opens a new one.

Once a problem is fitted to a frame, problem frames provides a way of methodically checking whether W, S ⊨ R holds and how this relationship may be broken. Here are some standard concerns that may be applied to illicit security vulnerabilities in the subproblem:

Identity Concern in Customers: Can the software discriminate one customer from another? If someone turns up and say s/he is an existing customer, or provide some details purporting to be an existing customer, can the software determine whether the claim is true? The answer is clear `no'. This environmental assumption is too weak. We may strengthen it by saying that the manager should check that a customer is who they say they. Again, this is not foolproof, but may be the best one can do. This assumption that the manager can really verify customer details is an important assumption in our proof obligation.

Credibility Concern in Managers: Will the manager always input the customer details faithfully? Can the manager act maliciously by fraudulently transferring money from a customer’s account to their personal accounts or those of their associates? Although it may be impossible or very difficult for the software to prevent such malicious actions, it is important for a requirements analyst to be aware that this is possible.

Interception Concern: All connections between the customers, managers, software and the database may be intercepted. For example if the customer has subscribed to online banking, there is a possibility that information sent or received from the bank server may be intercepted by attackers and replayed later. The analysis of this problem needs to take into account the possibility that this may occur and identify appropriate solutions to address this concern.

(ii) Evaluation of Security Pattern Support in Problem Frames

Problem Frames are a way of structuring software development problems, and concerns associated with the frames raise important security and other dependability issues. In this subsection we evaluate the extent to which problem frames support the representation of security concerns using the generic characterisation of security patterns presented in section 2.

Problem: The security goal analysed through a given problem frame depends on the concerns that are associated with that problem frame. For example the structuring of the bank account opening example as a workpiece raises confidentiality and accountability concerns of bank customer and managers, respectively. This illustrates that the structuring of software problems using problem frames leads to the identification and analysis of specific security concerns associated with the given frame/pattern. Therefore, although the intention of problem frames is classification of recurring software development problems, these patterns can be used to elicit associated security goals. It is also worth noting that even though security concerns can be identified from generic frame concerns, such identification of security goals depends on the specific problem represented in the problem frame.

Context in General: Explicit modelling of physical context and phenomena shared between domains is one of the core characteristics of problem frames. The context may represent the assets that may be harmed and domains that may interact with the assets. The domains interacting with the assets may either do so in good faith or with intent to cause harm. For example the bank manager may edit a customer account with the intent of serving the needs of the client or may act maliciously by fraudulently transferring money from the account. By explicitly showing the context in which an application will operate and relationships between domains, problem frames enables requirement analyst to assess potential threats that may arise.

Context for Attack: Potential attacks may be identified through frame concerns. However, there is no explicit risk analysis; that is, assessment of the probability of an attack occurring and the extent of the resulting damage that may be incurred. Hence frames concerns are sufficient for identifying potential attack, such as how do we authenticate the customer in order to ascertain that he is who he claims to be and is the legitimate owner of the bank account. In this respect, frames concerns allow requirements analysts to ask “what if” questions about the behaviour of the domains in a problem structure with respect to violation of security goals. Although, frame concerns may help surface potential attacks on assets, they are not detailed enough to explain scenarios

by which a security goal may be violated. This may be achieved by elaborating the identified security concerns with scenario-based approaches such as misuse cases [2, 44].

Context for Assets: As earlier stated, problem frames provide explicit modelling of the context in which a system will operate. This includes the assets that may be harmed and their potential attackers. It is worth mentioning that, initially, in a problem frame these assets and attackers are modelled as domains. It is only when security issues are taken into account that these domains can be classified as assets or potential attackers. For example, the customer in the bank account opening example is a domain but upon consideration of security concerns, s/he can also be a considered as a potential malicious customer who can commit identity fraud. Similarly, when security concerns are considered the bank account becomes an asset to be protected against harm from malicious customers and bank managers.

Context for Harm to Assets: Problem frames do not provide explicit modelling for harms to assets. It is possible, though to describe such harm as a security requirement in a different problem description once vulnerability has been identified.

Forces: Although there is no explicit support for trade-off analysis in problem frames, alternative solutions can be evaluated by re-analysing a problem with different context as demonstrated in the work on context-awareness [33]. The selection of a problem pattern depends on the characteristics of the problem at hand.

Solution: The emphasis in the problem frames approach is on structures of problems rather than solutions. In a problem frame, the solution is modelled as an abstract machine without going into details about its dynamic behaviour. In order to model the dynamic behaviour of a machine, problem frames is often complemented with behavioural languages such as the Event Calculus [37].

Consequences: The consequences for applying a given problem frame for analyzing a security problem can be evaluated through the concerns associated with the problem pattern. An adequacy argument is then used to demonstrate that the behaviour of the security solution (S) in the problem context (W) satisfies the security requirements.

Table 4.3.1 Evaluation of Support for Security Patterns in Problem Frames

Problem	Context				Forces	Solution	Consequences
	General	Attack	Assets	Harms to Assets			
YES	YES	YES	YES	NO	YES	NO	YES

The classification and structuring of problems with problem frames provides a systematic way of capturing recurring software development problems. The frames also document standard concerns that should be addressed when a problem that fits a given pattern is encountered. Identification and reasoning about the concerns of each frame provides a means to identifying potential security issues that need to be addressed for a given problem. Table 4.3.1 presents a summary of the evaluation of the support for security patterns in problem frames based on the discussion above.

4.3.2 Abuse Frames

(i) Overview of Abuse Frames

Lin et al. [21, 22] proposed abuse frames, an approach to analysing security problems in order to determine security vulnerabilities. This approach is based on Jackson's problem frames approach to structuring and analysing software development problems [17]. While problem frames are aimed at analysing the requirements to be satisfied, in contrast, abuse frames are based on the notion of an anti-requirement. An anti-requirement is the requirement of a malicious user that can subvert an existing requirement (similar to the concept of an anti-goal [39]).

Abuse frames represent the notion of a security threat imposed by malicious users and a means for bounding the scope of security problems in order to provide early focus for security threat analysis. Binding the scope of a security problem makes it possible to describe it more explicitly and precisely. Such explicit and precise descriptions facilitate the identification and analysis of threats, which in turn drive the elicitation and elaboration of security requirements.

(ii) Security Patterns in Abuse Frames

Abused frames are based on the notion of an anti-requirement. An anti-requirement defines a set of undesirable phenomena imposed by a malicious user that ultimately cause the system to reach a state that is inconsistent with its requirement. The representation of security analysis patterns in abuse frames share the

same notion as problem frames. However, in abuse frame frames the domains are associated with a different meaning. An abuse frame consists of three domains: *vulnerability machine*, *asset*, and *malicious user*; as shown in Figure 4.3.2a. Phenomenon P1 represents undesirable effects on the asset resulting from an attack and phenomenon P3 represents abuse actions sent by the vulnerability machine to the asset on behalf of the malicious user. Similarly, phenomenon P4 represents interaction between the malicious user and the vulnerable machine during an attack.

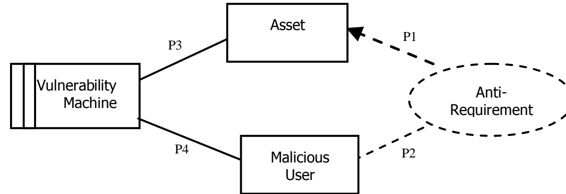


Figure 4.3.2a Generic Abuse Frame describing a threat

A vulnerability machine domain describes the behaviour that a malicious user exploits to make an attack possible. The asset domain represents the asset that will be harmed if an attack succeeds, while the malicious user domain represents a potential attacker. Similar to problems frames, each abuse frame is associated with a set of abuse frame concerns which need to be addressed to minimise possibility of an attack from being successful. Figure 4.3.2b is an abuse frame describing a possible attack on the bank account. The attack scenario described is one where a malicious bank manager transfers funds from an account without authorisation for the account owner.

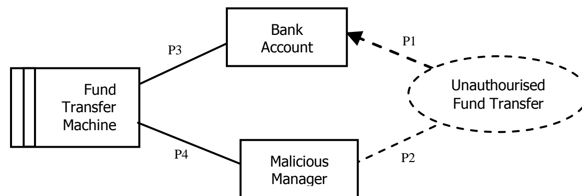


Figure 4.3.2b. Abuse Frame describing possible attack on a bank account

(iii) Evaluation of Patterns Support in Abuse Frames

Support for patterns in abuse frames is very similar to problem frames. The difference is that abuse frames have specific constructs for security analysis, while problem frames are generic patterns of software development problem. Table 4.3.2 presents a summary of the evaluation of security pattern support in abuse frames.

Table 4.3.2 Evaluation of Support for Security Patterns in Abuse Frames

Problem	Context				Forces	Solution	Consequences
	General	Attack	Assets	Harms to Assets			
YES	YES	YES	YES	YES	YES	NO	YES

As shown in the table, abuse frames provide explicit support for modelling potential attackers, assets, and harms to assets. By modelling the behaviour of an attacker, the properties assets, and the specification of a vulnerable machine that an attacker could use to harm assets; abuse frames provides systematic means of eliciting security requirements.

5. COMPARING MODELLING APPROACHES

Table 5.1 presents a summary of the evaluation results presented in section 4. Our evaluation results suggest that approaches to security analyses are similar in their capabilities to capturing and modelling security analysis patterns. The differences between the features of the approaches are dependent on the different concerns they were originally meant to address. This suitability of each security analysis approach to specific security concerns makes it possible for the approaches to complement each other in a number of ways.

Table 5.1 Comparison of Modelling Approaches

Modelling Approaches		Problem	Context				Forces	Solution	Consequences
			General	Attack	Assets	Harms to Assets			
Design-Oriented	UML	NO	YES	YES	YES	NO	NO	YES	NO
	SecureUML	NO	YES	NO	YES	NO	NO	YES	NO
	UMLSec	NO	YES	YES	NO	YES	YES	YES	YES
	Misuse Cases	YES	YES	YES	NO	YES	NO	YES	YES
Goal-Oriented	Secure i*	YES	YES	YES	YES	NO	YES	NO	YES
	Secure Tropos	YES	YES	YES	YES	YES	YES	YES	YES
	KAOS	YES	YES	YES	YES	NO	NO	YES	YES
Problem-Oriented	Problem Frames	YES	YES	YES	YES	NO	NO	NO	YES
	Abuse Frames	YES	YES	YES	YES	YES	YES	NO	YES

Problem-oriented and goals oriented approaches document problem structures in the problem space; while design approaches document solution patterns. Goal-based approaches are useful in documenting security goals at a higher-level of abstraction than problem-oriented approaches. Goal-based approaches are useful for capturing and refining security goals. Problem-oriented approaches further analyse the security goals structuring them into problem structures whose solutions can be identified with design approaches.

6. CONCLUSION AND FURTHER WORK

Security remains a key challenge in the development of software systems and the goal of developing secure software systems has remained an area of active research. Research in security engineering has resulted in the realization that documenting recurring security problems and their solutions as security patterns is an important advancement as it allows software designers with little knowledge of security to build secure systems. When a designer encounters a security problem that match a given pattern, they can reuse the solution part of the pattern or use the pattern to guide them in finding a solution to the problem at hand. In this chapter we have reviewed approaches to security analysis. Our review focused on evaluating the capabilities of these approaches to supporting security analysis patterns and is based on a set of evaluation criteria for characterising security patterns.

Although the approaches reviewed were originally aimed at addressing specific security concerns our evaluation results suggest that these approaches are, to a large extent, overwhelmingly similar in their capabilities to capturing and modelling security analysis patterns. The minor differences in their capabilities to modelling security patterns seems to imply that these approaches complement each other in a number of ways. Problem-oriented approaches document problem structures in the problem space; while design approaches document solution patterns. Goal-based approaches are useful in capturing and refining security goals at a higher-level of abstraction than problem-oriented approaches. On the other hand, the explicitness of context modelling in problem-oriented approaches make them more suited to further analyses of security goals, structuring them (security goals) into problem structures whose solutions can be identified with design approaches. This systematic structuring of security problems could potentially result in more clearly defined structures of common security goals such as confidentiality, integrity, etc, into generic problem structures that can be instantiated for analysing security problems in different application domains.

Although our conclusion is based on the evaluation representation of a single security patterns with different languages, we believe that our results may be generalized to other security patterns. Part of our future work is validating the extent to which the conclusions we have draw here can be generalized.

References

1. Alexander, I. *Initial industrial experience of misuse cases in trade-off analysis*. in *Proceedings of IEEE Joint International Conference on Requirements Engineering*. 2002.
2. Alexander, I., *Misuse cases: use cases with hostile intent.* IEEE Software, 2003. **20**(1): p. 58-66.
3. ANSI/INCITS, *Information Technology - Role Based Access Control*, in *InterNational Committee for Information Technology Standards*. 2004.

4. Arlow, J., *Use cases, UML visual modelling and the trivialisation of business requirements.* Requirements Engineering, 1998. **3**(2): p. 150-152.
5. Bresciani, P., A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos, *Tropos: An Agent-Oriented Software Development Methodology.* Autonomous Agents and Multi-Agent Systems, 2004. **8**(3): p. 203-236.
6. Castroa, J., M. Kolp, and J. Mylopoulos, *Towards requirements-driven information systems engineering: the Tropos project.* Information Systems, 2002. **27**(6): p. 365-389.
7. Darimont, R. and A.v. Lamsweerde, *Formal refinement patterns for goal-driven requirements elaboration.* ACM SIGSOFT Software Engineering Notes, 1996. **21**(6): p. 179-190.
8. Fernandez, E., G. Pernul, and M. Larrondo-Petrie, *Patterns and Pattern Diagrams for Access Control*, in *Trust, Privacy and Security in Digital Business*. 2008. p. 38-47.
9. Fernandez, E.B. and R. Pan. *A pattern language for security models*. in *Proceedings of 8th Conference on Pattern Languages of Programs (PLoP)*. 2001. Monticello, Illinois, USA.
10. Fernández-Medina, E., J. Jurjens, J. Trujillo, and S. Jajodia, *Model-Driven Development for secure information systems.* Information and Software Technology, 2009. **51**(5): p. 809-814.
11. Ferraiolo, D.F., R. Sandhu, S. Gavrila, D.R. Kuhn, and R. Chandramouli, *Proposed NIST standard for role-based access control.* ACM Transactions on Information and System Security, 2001. **4**(3): p. 224-274.
12. Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994, New Jersey: Addison-Wesley. 385.
13. Giorgini, P., J. Mylopoulos, E. Nicchiarelli, and R. Sebastiani, *Reasoning with Goal Models*, in *Proceedings of the 21st International Conference on Conceptual Modeling*. 2002, Springer-Verlag. p. 167-181.
14. Hall, J.G., L. Rapanotti, and M. Jackson. *Problem Oriented Software Engineering: A design-theoretic framework for software engineering*. in *5th IEEE International Conference on Software Engineering and Formal Methods*. 2007.
15. Hall, J.G., L. Rapanotti, and M.A. Jackson, *Problem Oriented Software Engineering: Solving the Package Router Control Problem.* IEEE Transactions on Software Engineering, 2008. **34**(2): p. 226-241.
16. Hoare, C.A.R., *Communicating sequential processes.* Commununications of the ACM, 1983. **26**(1): p. 100-106.
17. Jackson, M., *Problem frames : analysing and structuring software development problems*. ACM Press. 2001, Harlow: Addison-Wesley, 2001.
18. Jackson, M., *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices*. 1995, London, United Kingdom: Addison-Wesley. 228.
19. Jacobson, I., *Object Oriented Software Engineering: A Use Case Driven Approach*. 1992: Addison-Wesley Professional. 552.
20. Jurjens, J., *Secure Systems Development with UML*. 2004, Heidelberg, German: Springer-Verlag. 316.
21. Lin, L., B. Nuseibeh, D. Ince, and M. Jackson. *Using abuse frames to bound the scope of security problems*. in *Proceedings of 12th IEEE International Requirements Engineering Conference*. 2004.
22. Lin, L., B. Nuseibeh, D. Ince, M. Jackson, and J. Moffett. *Introducing abuse frames for analysing security requirements*. in *Proceedings of 11th IEEE International Requirements Engineering Conference*. 2003.
23. Liu, L., E. Yu, and J. Mylopoulos. *Security and privacy requirements analysis within a social setting*. in *Proceedings of 11th IEEE International Requirements Engineering Conference*. 2003.
24. Lodderstedt, T., D. Basin, and J. Doser, *SecureUML: A UML-Based Modeling Language for Model-Driven Security*, in *«UML» 2002: The Unified Modeling Language*. 2002. p. 426-441.
25. Manna, Z. and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. 1992: Springer Verlag. 448.
26. Matulevicius, R., N. Mayer, H. Mouratidis, E. Dubois, P. Heymans, and N. Genon, *Adapting Secure Tropos for Security Risk Management in the Early Phases of Information Systems Development*, in *Proceedings of the 20th international conference on Advanced Information Systems Engineering*. 2008, Springer-Verlag: Montpellier, France. p. 541-555.
27. Mouratidis, H. (2004), *A Security Oriented Approach in the Development of Multiagent Systems: Applied to the Management of the Health and Social Care Needs of Older People In England:* PhD Thesis Thesis, University of Sheffield: Sheffield, UK
28. Mouratidis, H. and P. Giorgini, *Secure Tropos: A Security-Oriented Extension of the Tropos methodology.* International Journal of Software Engineering and Knowledge Engineering, 2007. **27**(2): p. 285-309.
29. Mouratidis, H., P. Giorgini, and G. Manson, *When security meets software engineering: a case of modelling secure information systems.* Information Systems, 2005. **30**(8): p. 609-629.
30. Mouratidis, H., M. Weiss, and P. Giorgini, *Modelling Secure Systems Using An Agent Oriented Approach and Security Patterns.* International Journal of Software Engineering and Knowledge Engineering, 2006. **16**(3): p. 471-498.
31. OMG, *UML Profile for Patterns Specification*, in *UML Profile for Enterprise Distributed Object Computing (EDOC) specification*. 2004.
32. OMG, *Unified Modeling Language (UML)*. 2009.
33. Salifu, M., Y. Yu, and B. Nuseibeh. *Specifying Monitoring and Switching Problems in Context*. in *Proceedings of the 15th IEEE International Conference in Requirements Engineering (RE '07)*. 2007. New Delhi, India.

34. Sandhu, R.S., E.J. Coyne, H.L. Feinstein, and C.E. Youman, *Role-based access control models.* Computer, 1996. **29**(2): p. 38-47.
35. Schumacher, M., E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad, *Security Patterns: Integrating Security and Systems Engineering*. Software Design Patterns. 2005, West Sussex, England: John Wiley & Sons. 600.
36. Sebastiani, R., P. Giorgini, and J. Mylopoulos, *Simple and Minimum-Cost Satisfiability for Goal Models*, in *Advanced Information Systems Engineering*. 2004. p. 20-35.
37. Shanahan, M., *The Event Calculus Explained*, in *Artificial Intelligence Today: Recent Trends and Developments*. 1999, Springer: Berlin / Heidelberg. p. 409-430.
38. Sindre, G. and A.L. Opdahl, *Eliciting security requirements with misuse cases.* Requirements Engineering, 2005. **10**(1): p. 34-44.
39. van Lamsweerde, A. *Elaborating security requirements by construction of intentional anti-models*. in *26th International Conference on Software Engineering*. 2004.
40. van Lamsweerde, A. *Elaborating security requirements by construction of intentional anti-models*. in *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004).* 2004.
41. van Lamsweerde, A., R. Darimont, and P. Massonet. *Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt*. in *Proceedings of the 2nd IEEE International Symposium on Requirements Engineering*. 1995.
42. Wang, Y., S.A. McIlraith, Y. Yu, and J. Mylopoulos, *An automated approach to monitoring and diagnosing requirements*, in *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*. 2007, ACM: Atlanta, Georgia, USA. p. 293-302.
43. Weidenhaupt, K., K. Pohl, M. Jarke, and P. Haumer. *Scenario usage in system development: a report on current practice*. in *Proceedings of the 3rd International Conference on Requirements Engineering*. 1998.
44. Whittle, J., D. Wijesekera, and M. Hartong. *Executable misuse cases for modeling security concerns*. in *ACM/IEEE 30th International Conference on Software Engineering*. 2008.
45. Yoshioka, N., H. Washizaki, and K. Maruyama, *A survey on security patterns.* Progress in Informatics, 2008(5): p. 35-47.
46. Yu, E.S.K. *Towards modelling and reasoning support for early-phase requirements engineering*. in *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*. 1997.
47. Yu, Y., H. Kaiya, H. Washizaki, Y. Xiong, Z. Hu, and N. Yoshioka, *Enforcing a security pattern in stakeholder goal models*, in *Proceedings of the 4th ACM Workshop on Quality of Protection*. 2008, ACM: Alexandria, Virginia, USA. p. 9-14.
48. Mayer, N., Heymans, P., and Matulevicius, R., Design of a Modelling Language for Information System Security Risk Management, In proceedings of the 1st International Conferences on Research Challenges in Information Science (RCIS), 2007, pp.121-131.
49. Cabot, J., and Zannone, N., Towards an Integrated Framework for Model-driven Security Engineering, in Proceedings of the Proceedings of the Workshop on Modeling Security (MODSEC08), 2008, CEUR Workshop Proceedings.