Describing Early Security Requirements using Use Case Maps

Jameleddine Hassine[1] and Abdelwahab Hamou-Lhadj[2]

[1] Department of Information and Computer Science
King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia
jhassine@kfupm.edu.sa

[2] Electrical and Computer Engineering Department
Concordia University, Montréal, Canada
abdelw@ece.concordia.ca

Abstract. Non-functional requirements (NFR), such as availability, usability, performance, and security are often crucial in producing a satisfactory software product. Therefore, these non-functional requirements should be addressed as early as possible in the software development life cycle. Contrary to other non-functional requirements, such as usability and performance, security concerns are often postponed at the very end of the design process. As a result, security requirements have to be tailored into an existing design, leading to serious design challenges that usually translate into a software vulnerabilities. Security architectural tactics describe security design measures in a very general, abstract, and implementation-independent way. In this paper, we present a novel approach to describe high-level security requirements using the Use Case Maps (UCM) language of the ITU-T User Requirements Notation (URN) standard. The proposed approach is based on a mapping of the well-known security tactics to UCM models. The resulting security extensions are described using a metamodel and are implemented within the jUCMNav tool. We illustrate our approach using a UCM scenario describing the modification of consultants pay rates.

1 Introduction

In the early stages of common development processes, system functionalities are defined in terms of informal requirements and visual descriptions. Scenarios are a well established approach to describe functional requirements, uncovering hidden requirements and trade-offs, as well as validating and verifying requirements. The Use Case Maps (UCM) language, part of the ITU-T User Requirements Notation (URN) standard [1], is a high-level visual scenario-based modeling language that has raised a lot of interest in recent years within the software requirements community. Use Case Maps can be used to capture and integrate functional requirements in terms of causal scenarios representing behavioral aspects at a high level of abstraction, and to provide the stakeholders with guidance and reasoning about the system-wide architecture and behavior.

Non-functional attributes such as availability, performance, and security are often overlooked during the initial system design. Clements and Northrup [2] have suggested that whether or not a system will be able to exhibit its required quality attributes (NFRs) is largely determined by the selected architecture. Hence, system architecture should address both functional and non-functional requirements.

In order to solve commonly occurring problems in software architecture, architectural patterns were introduced as a well-known reusable solution within a given context [3]. Despite their popularity, architectural patterns suffer from a number of criticisms and deficiencies. One of these weaknesses is that an architectural pattern usually address multiple quality attributes at once [4]. To overcome this weakness, the notion of tactics has been proposed by Bass et al. [5] as *architectural building blocks* of architectural patterns in order to achieve quality attributes, such as availability, safety, and security. As with architectural patterns, architectural tactics emerge from practice through empirical experiments and observations.

It is well-known that software flaws are very expensive when found later in the system development life-cycle. More specifically, security vulnerabilities left in the released software may be catastrophic. Hence, there is a need to consider security from the early stages of the software systems development.

The widespread interest in security modeling and analysis techniques, constitutes the major motivation of this paper. We, in particular, focus on the need to incorporate security aspects at the very early stages of system development. This work builds upon and extends our previous work on describing and assessing availability requirements using the Use Case Maps language [6,7,8,9] and the Aspect-Oriented Use Case Maps (AoUCM) [10]. This paper serves the following purposes:

- It adopts the security tactics introduced by Bass et al. [11] as a basis for extending the Use Case Maps language with security-related requirements.
- It describes a set of UCM-based security extensions using a metamodel. These extensions are implemented using the jUCMNav tool through metadata mechanism.
- It extends our ongoing research towards the construction of a UCM-based framework for the description and analysis of non-functional requirements in the early stages of system development life cycle.

The remainder of this paper is organized as follows. The next section provides an overview of system security requirements. In Sect. 3, we present and discuss the proposed UCM-based security annotations. An example of a UCM scenario describing the modification of consultants pay rates is presented in Sect. 5. Section 6 discusses related work, the benefits and the shortcomings of our proposed approach. Finally, conclusions are drawn in Sect. 7.

2 Security Requirements

In the ITU-T recommendation E.800 [12], the term '*security*' is used in the sense of minimizing the vulnerabilities of assets and resources. An asset is defined as 'anything of value', while a vulnerability is defined as 'any weakness that could be exploited to violate a system or its data'. ITU-T in its recommendation X.1051 [13] defines Information security as security preservation of confidentiality, integrity, and availability of information.

Security can be characterized in terms of confidentiality (i.e., no unauthorized subject can access the content of a message), integrity (e.g., message content cannot be altered), and availability (i.e., system available for legitimate use). Other characteristics, such as authentication (checking the identity of a client), authorization (checking whether a client might invoke a certain operation), and non-repudiation (which refers to the accountability of the communicating parties), are used to support security.

Bass et al. [5] have provided a comprehensive categorization of security tactics) based on whether they address the detection of, the resistance to, and the recovery from attacks. A refined hierarchy of security tactics has been presented later in [11] by adding an additional category of tactics to deal with reacting to attacks and by refining the existing categories. Figure 1 illustrates the four classes of tactics, where the directed arrows show refinement relationships and each element represents an individual tactic:

1. **Detect Attacks** category consists of four tactics:
   - *Detect intrusion* tactic refers to the ability to recognize typical attack pattern trough monitoring and analyzing both user and system/network activities. The detection intrusion tactic can be realized, for example, using a comparison of network traffic (inbound and outbound) or service requests with a set of signatures of known malicious patterns, e.g., TCP flags, payload sizes, source or destination address, port number, etc.
   - *Detect service denial* tactic refers to the ability to detect attempts to make a machine or network resource unavailable (temporarily or indefinitely) to its intended users. It can be realized, for instance, by comparing the pattern/signature of the incoming network traffic with historic profiles of known denial of service attacks.
   - *Verify message integrity* tactic employs techniques such as checksums or hash values to check the integrity of messages, resource files, and configuration files.
   - *Detect message delay* tactic is intended to detect potential man-in-the-middle (MITM) attacks, where the attacker secretly is intercepting and possibly altering the communication between two parties who believe they are directly communicating with each other. This tactic can be realized, for instance, by examining the latency of the exchanged messages.
2. **Resist Attacks** category is divided into eight tactics:
   - *Identify actors* tactic refers to the ability of identifying the source (e.g., user IDs, IP addresses, protocols, etc.) of any external input to the system.

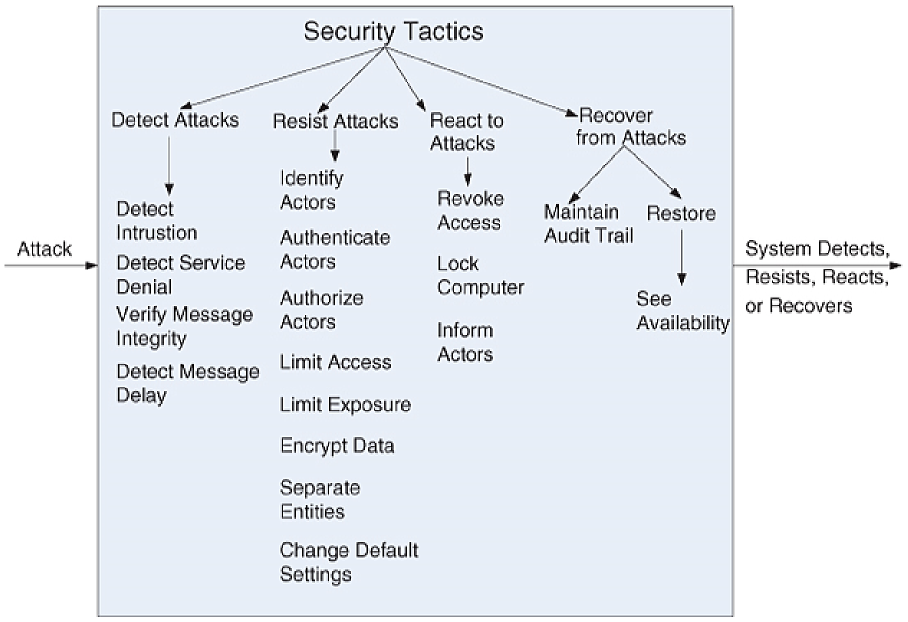


Fig. 1. Security Tactics [11]

– *Authenticate actors* tactic ensures that an actor (a user of a computer) is who he claims to be. It can be realized, for instance, by using passwords, digital certificates, and biometric identification.
– *Authorize actors* tactic ensures that only certain authenticated actors have access to a resource (data or services). It can be realized, for example, by specifying access control mechanisms.
– *Limit access* tactic aims to limit the access to resources such as network connections, memory, etc. It may be achieved by blocking a host, closing a port, or rejecting a protocol.
– *Limit exposure* tactic focuses on minimizing the attack surface. It does not proactively prevent attackers from causing harm, but tries to minimize the effect of damage. It may be achieved by having a limited number of access points for resources, data, or services.
– *Encrypt data* tactic provides extra protection to persistently maintained data beyond that available from authorization. Encryption offers protection (e.g., through VPN or SSL) for passing data over publicly accessible communication links.
– *Separate entities* tactic ensures the separation of different entities within a system (e.g., different servers attached to different networks). Sensitive data is usually separated from nonsensitive data to reduce the attack possibilities from those who have access to nonsensitive data.

  - *Change default settings* tactic forces the user to change default settings, which will prevent attackers from gaining access to the system through settings that are, generally, publicly available.
3. **React to Attacks** category consists of three tactics:
   - *Revoke access* tactic ensures that access to sensitive resources is limited if an attack is underway.
   - *Lock computer* tactic ensures that a limited access is granted to potentially malicious parties, for example, in case of repeated failed login attempts.
   - *Inform actors* tactic refers to the ability to notify intervening parties in case of an ongoing attack.
4. **Recover from Attacks** tactics are divided into:
   - *Service restoration* tactic ensures the recovery of the system after an attack. It may be realized through redundant hardware. Availability tactics can be deployed to achieve service restoration.
   - *Maintain audit trail* tactic is used to trace the actions of and to identify an attacker.

In this research, we adopt these security tactics introduced by Bass et al. [11] as a basis for extending the Use Case Maps language [1] with security annotations. These tactics have been proven in practice for a broad applicability in different industrial domains.

3 Security Modeling in Use Case Maps

The URN standard [1] offers mechanisms in order to support the profiling of the language to a particular domain. One such mechanism is *Metadata*, which are name-value pairs that can be used to tag any URN specification or its model elements, similar to stereotypes in UML. Metadata instances provide modelers with a way to attach user-defined named values to most elements found in a URN specification, hence providing an extensible semantics to URN. A metadata is described using a name (string) and a value (string) of the URN metadata information instance. In this paper, we propose to implement our security extensions within *jUCMNav* [14], the most comprehensive URN tool available to date, using *metadata* feature.

In what follows, we adopt the security tactics introduced by Bass et al. [11] as a basis for extending the Use Case Maps language with security annotations.

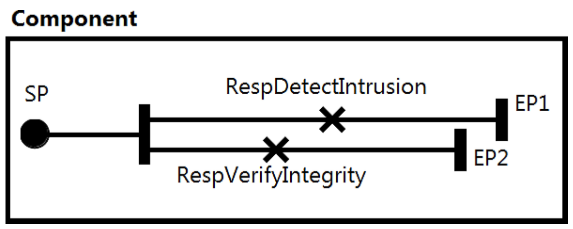
3.1 UCM Attack Detection Modeling

The specification of attack detection mechanisms is a key factor in implementing any security strategy. They are modeled and handled at the scenario path level, by associating the type of the deployed detection method with UCM responsibilities along the execution path. The security requirements of a responsibility can be modeled using two metadata attributes:

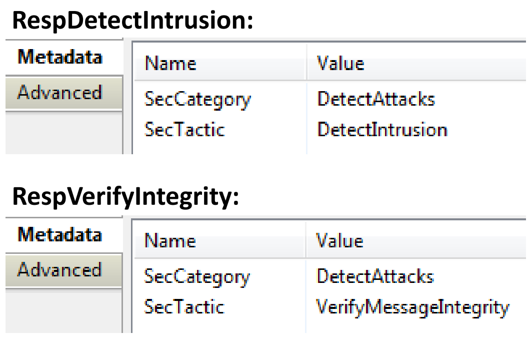
1. *SecCategory:* Specifies the security category, if any, that the responsibility is implementing. In the case of attack detection, it is specified as "*DetectAttacks*".
2. *SecTactic:* Denotes the type of the deployed security tactic. This attribute may take one of the following four values: *DetectIntrusion*, *DetectServiceDenial*, *VerifyMessageIntegrity*, and *DetectMessageDelay*, in case the value *DetectAttacks* is selected for the *SecCategory*.

The realization of the *DetectAttacks* tactic is assured by the definition of these two metadata attributes. A detailed definition of these attributes and their possible values is described as part of the UCM security metamodel in Section 4.

Figure 2 illustrates a UCM having two parallel (implemented using a UCM AND-fork constructor) responsibilities (i.e., *RespDetectIntrusion* and *RespVerifiyIntegrity*) implementing two attack detection (i.e., *DetectAttacks* category) tactics, *DetectIntrusion* and *VerifyMessageIntegrity*, respectively.



(a) UCM Attack Detection Modeling


RespDetectIntrusion:

Metadata | Advanced

Name	Value
SecCategory	DetectAttacks
SecTactic	DetectIntrusion

RespVerifyIntegrity:

Metadata | Advanced

Name	Value
SecCategory	DetectAttacks
SecTactic	VerifyMessageIntegrity

(b) Attack Detection Metadata Attributes

Fig. 2. UCM Attack Detection Modeling

Dealing with an attack (e.g., resist, react to, or recover from an attack) after detection is modeled using failure scenario paths as described in the following sections.

3.2 UCM Attack Resistance, Reaction, and Recovery Modeling

Given the fact that we have adequate detection mechanisms in place to detect an attack, a system may be able to resist the ongoing attack. In the case of an unsuccessful resistance, a system may be able to react to the attack. Finally, the system may be compromised (e.g., resources compromised, lost data, etc.), if the system has been compromised (e.g., resources compromised, lost data, etc.). In such a case, the system shall be able to recover from the attack.

The realization of the resistance, reaction, and recovery tactics are assured by:

– The definition of metadata attributes, attached to responsibilities, targeting the resistance (i.e., *ResistAttack*), reaction (i.e., *ReackAttack*), and recovery (i.e., *RecoverAttack*) categories. Similarly to the attack detection modeling, the resistance, reaction, and recovery can be modeled using *SecCategory* and *SecTactic* metadata attributes.
– Defining a hierarchical (using UCM stubs) structure of cascading failure scenario paths. A failure path starts with a failure start point (indicated by the F inside it) and a guarding condition (see Fig. 3). The guard condition can be initialized as part of a scenario definition (i.e., scenario triggering condition) or can be modified as part of a the responsibility expression.

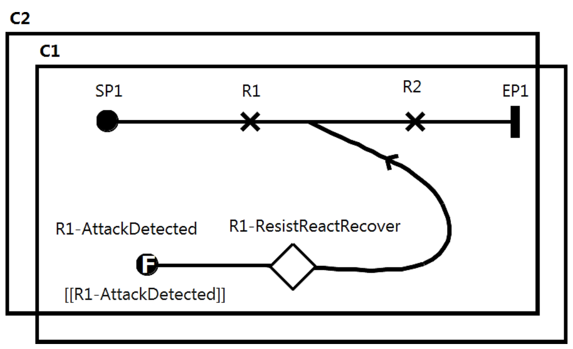
Figure 3 illustrates a generic UCM map with a main scenario starting at start point SP1 and executing responsibilities R1 and R2. Responsibility R1 implements the *DetectIntrusion* tactic, part of the *DetectAttack* category. A successful detection of an intrusion triggers a failure scenario path, by setting the failure guard *R1-AttackDetected* to true. Responsibility *R1* may execute the following code:

```
if (R1_AttDetected)
    R1-AttackDetected := true;
else
    R1-AttackDetected := false;
```

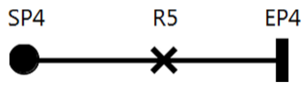
where R1_AttDetected is a Boolean variable that can be initialized as part of a scenario definition.

However, the addition of metadata to responsibilities requires a change to the standard UCM traversal mechanism because a path may have to be stopped at a responsibility and continued at a failure start point.

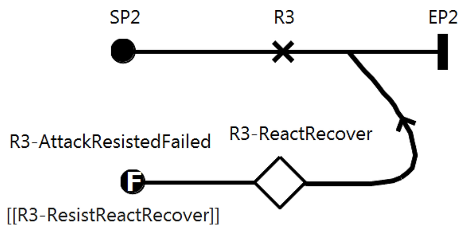
The execution of the failure path leads to the execution of a plugin embedded in the static stub *R1-ResistReactRecover* (see Fig 3(c)) starting at start point SP2. Responsibility R3 realizes the *LimitAccess* attack resistance tactic. An unsuccessful resistance to the intrusion attack would trigger a failure path that starts at failure start point *R3-AttackResistedFailed* and executes the *R3-ReactRecover* stub (Fig. 3(d) illustrates its corresponding plugin). Responsibility R4 models the *RevokeAccess* tactic, part of the *ReactAttacks* category. A failure to react to the attack triggers a failure path that executes the *R4-Recover*



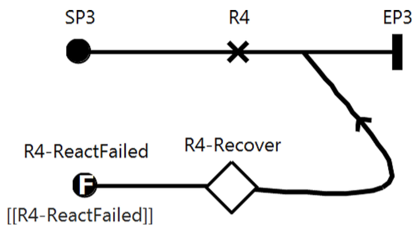
(a) UCM failure scenario path



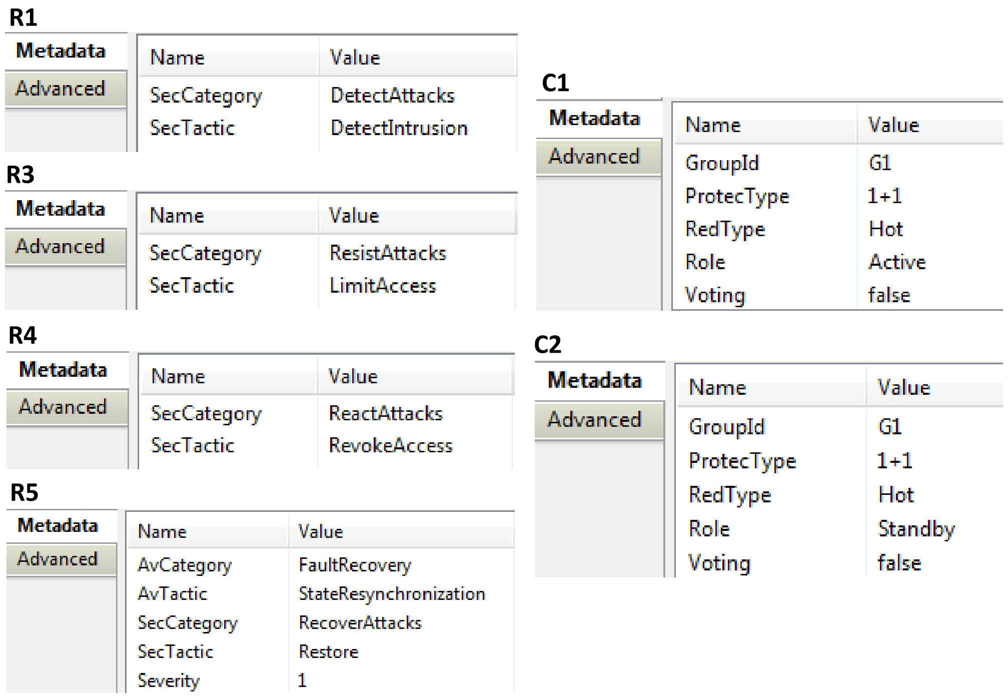
(b) R4-Recover plugin



(c) R1-ResistReactRecover plugin



(d) R3-ReactRecover plugin



R1

Name	Value
SecCategory	DetectAttacks
SecTactic	DetectIntrusion

R3

Name	Value
SecCategory	ResistAttacks
SecTactic	LimitAccess

R4

Name	Value
SecCategory	ReactAttacks
SecTactic	RevokeAccess

R5

Name	Value
AvCategory	FaultRecovery
AvTactic	StateResynchronization
SecCategory	RecoverAttacks
SecTactic	Restore
Severity	1

C1

Name	Value
GroupId	G1
ProtecType	1+1
RedType	Hot
Role	Active
Voting	false

C2

Name	Value
GroupId	G1
ProtecType	1+1
RedType	Hot
Role	Standby
Voting	false

(e) R1, R3, R4, R5, C1, and C2 Metadata Attributes

Fig. 3. UCM Modeling of attack resistance, reaction, and recovery

plugin. Responsibility *R5* implements the *Restore* tactic, part of the *RecoverAttacks* category. The *Restore* tactic is refined using availability tactics (see Fig. 1). Recovery focuses mainly on redundancy modeling in order to keep the system available. The UCM of Fig. 3(a) illustrates two components C1 and C2 participating in a 1+1 hot redundancy configuration. *C1* is in active role, while *C2* is in standby role. None of these two components is taking part in a voting activity (i.e., Voting : *false*). For a detailed description of the UCM-based availability tactics, interested readers are referred to [8].

Figure 3(e) shows the metadata corresponding to responsibilities R1, R3, R4, R5, and components C1 and C2.

It is worth noting that a system might not implement all categories of tactics (i.e., resist, react, and recover categories). In such a case, the UCM cascading hierarchy may be reduced to one or two levels only. The example in Sect. 5 illustrates such a case.

4 UCM Security-Enabled Metamodel

In this section, we describe our UCM-based security extensions using an abstract grammar metamodel. The concrete grammar metamodel, which includes metaclasses of the graphical layout of UCM elements, is not discussed in this paper since they have no semantic implications.

Figure 4 illustrates an excerpt of the UCM language core abstract metamodel augmented with security and availability concepts. *UCMspec* serves as a container for the UCM specification elements such as *Component* and *Responsibility.* Path-related (e.g., AND-Fork, OR-Fork, etc.) and plugin binding-related concepts are not shown because they do not impact our security and availability extensions.

Two new security-related enumeration metaclasses (shown in yellow) are introduced:

- **SecurityCategory:** Specifies the category of the tactic a responsibility is implementing (e.g., *DetectAttacks*, *ResistAttacks*, *ReactAttacks*, and *RecoverAttacks*).
- **SecurityTactic:** Specifies which tactic a responsibility is realizing (e.g., *DetectIntrusion*, *AuthenticateActors*, etc.)

An additional metaclass *ResponsibilitySecurity* (shown in yellow) is introduced to define the security attributes attached to a responsibility:

- **SecCategory** of type *SecurityCategory.*
- **SecTactic** of type *SecurityTactic.*

It is worth noting that one single responsibility may implement one security tactic only (as described using the 0..1 relationship multiplicity in the metamodel). Responsibilities shall be refined into multiple responsibilities when there is a need to realize more than one security tactic.

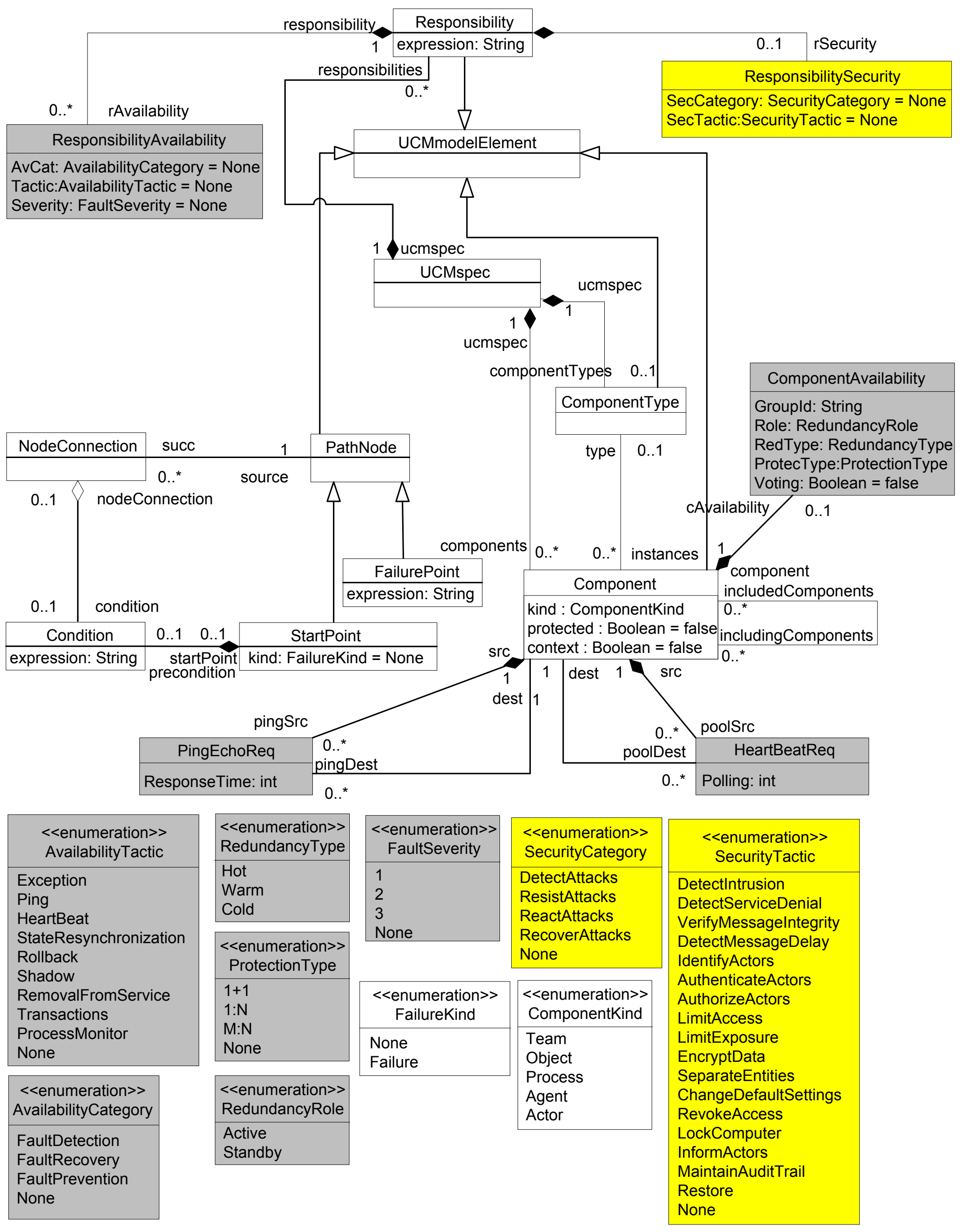


Fig. 4. Abstract Grammar: UCM Security-Enabled Metamodel

In addition, we reuse the existing set of availability tactics, defined in [8] to model availability requirements such as component redundancy, and fault detection, recovery, and prevention.

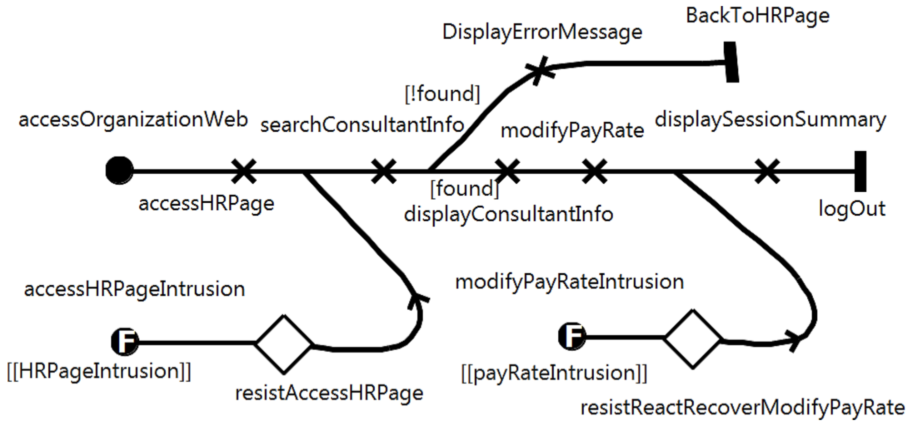
5 Illustrative Example: Modification of Consultant Pay Rate

In this section, we illustrate our proposed approach using a case study describing the modification of consultants pay rates. Such a critical task should be performed by an HR employee (having special privileges) from inside the organization (using the local intranet). The regular scenario (without security aspects), starts by accessing the organization local web page (i.e., responsibility *accessOrganizationWeb*), then accessing the HR web page (i.e., responsibility *accessHRPage*). In order to change a specific consultant pay rate (i.e., responsibility *modifyPayRate*), the operator should search for the consultant data (responsibility *searchConsultantInfo*) and if found he proceeds with the modification of the pay rate, otherwise an error message is displayed (i.e., responsibility *DisplayErrorMessage*). Finally, a summary of the actions performed during the session is displayed (i.e., responsibility *displaySessionSummary*).

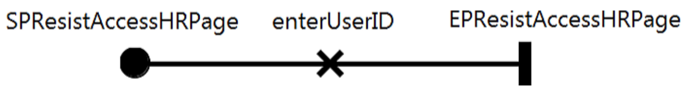
Security requirements are added to the regular scenario by attaching metadata attributes, describing security-related information, to the responsibilities, and by adding corresponding failure paths.

A potential attacker accessing the HR page from outside the organization (using an intermediate proxy) may result in some delay. The detection of such an intrusion can be achieved by attaching metadata attributes to responsibility *accessHRPage* (i.e., *SecTactic* = *DetectMessageDelay*) (see Fig. 6(a)). Once the attack is detected, a tentative to resist it is performed, using the failure path starting at failure start point *accessHRPageIntrusion*). In order to resist to the attack, the system tries to authenticate the user (i.e., responsibility *enterUserID* that realizes the *AuthenticateActors* tactic, see Fig. 6(c)). No further security related actions are taken (neither reaction nor recovery are modeled), in case the attack resistance fails.

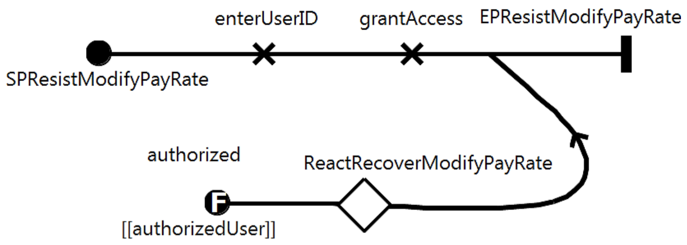
The modification of a consultant pay rate is a critical task and requires a protection against potential intrusions. This is achieved by attaching the *DetectIntrusion* tactic to the responsibility *modifyPayRate.* If a malicious intrusion is detected, it triggers a failure scenario path starting at failure start point *modifyPayRateIntrusion.* The resistance to the intrusion is realized using two responsibilities *enterUserID* and *grantAccess* realizing, the *AuthenticateActors* and *AuthorizeActors* tactics, respectively. An unsuccessful resistance to the intrusion, triggers a failure scenario path, starting at failure start point *authorized* and executing an attack reaction procedure. Figure 5(d), illustrates the plugin that corresponds to the static stub *ReactRecoverModifyPayRate.* The responsibility *denyAccess* realizes the *RevokeAccess* tactic, part of the *ReactAttacks* category. A failure to react to the intrusion, triggers a failure scenario path starting at failure start point *denyAccess-failed* and executes the plugin of the



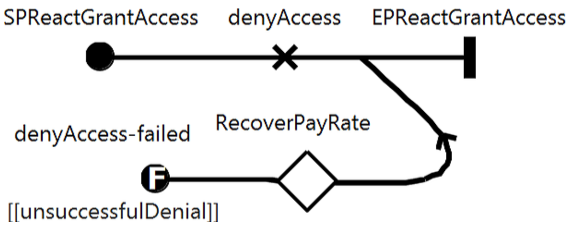
(a) Root Map



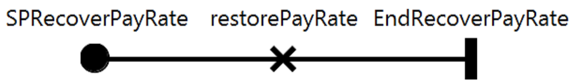
(b) resistAccessHRPage plugin



(c) resistReactRecoverModifyPayRate plugin

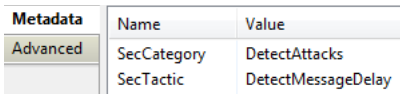


(d) ReactRecoverModifyPayRate plugin



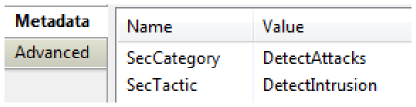
(e) RecoverPayRate plugin

Fig. 5. Modify Consultant Pay Rate Scenario



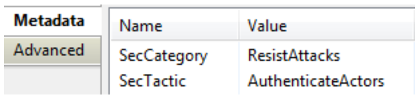
Name	Value
SecCategory	DetectAttacks
SecTactic	DetectMessageDelay

(a) accessHRPage metadata



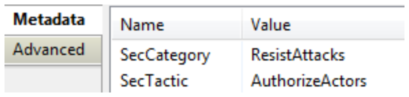
Name	Value
SecCategory	DetectAttacks
SecTactic	DetectIntrusion

(b) modifyPayRate metadata



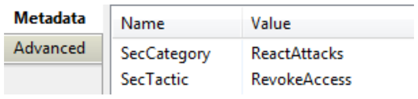
Name	Value
SecCategory	ResistAttacks
SecTactic	AuthenticateActors

(c) enterUserID metadata



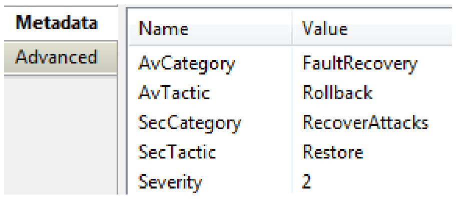
Name	Value
SecCategory	ResistAttacks
SecTactic	AuthorizeActors

(d) grantAccess metadata



Name	Value
SecCategory	ReactAttacks
SecTactic	RevokeAccess

(e) denyAccess metada



Name	Value
AvCategory	FaultRecovery
AvTactic	Rollback
SecCategory	RecoverAttacks
SecTactic	Restore
Severity	2

(f) restorePayRate metada

Fig. 6. Responsibilities Metadata Information

static stub *RecoverPayRate*. Responsibility *restorePayRate* realizes the *Restore* tactic, which also realizes the *Rollback* availability tactic (part of the *FaultRecovery* availability tactic).

6 Discussion

The need to consider security aspects during the early stages of the system development has been recognized by the requirements engineering community. Many techniques and methods have been proposed in the literature [15,16,17]. Misuse cases [16], abuse cases [15], and security use cases [17] are security-oriented variants of regular use cases.

Unlike regular use cases that describe normal interactions between an application and its users, misuse cases [16] and abuse cases [15] concentrate on interactions between the application and its misusers (i.e., potential attackers) who seek to violate its security requirements. These interactions are harmful to the system, one of the actors, or one of the stakeholders in the system. Security use cases [17] describe countermeasures intended to respond these attacks.

The most closely related work to ours is the one by Karpati et al. [18]. The authors have introduced the notion of *Misuse Case Maps (MUCM)* as a modeling technique that is the anti-behavioral complement to Use Case Maps, which is used to visualize how cyber attacks are performed in an architectural context. Karpati et al. [18] introduced a new set of symbols to visualize potential attack scenarios. These symbols are used to model exploit paths, vulnerable parts (points and responsibilities), misuser actions (using arrows specifying getting/putting/deleting/destroying components), etc. Our approach is different from the one in [18] with respect to two points:

- In our work, we view security requirements as assets and services that have to be protected against possible attacks. Hence, our goal is to guard functional behavior against potential threats. This is achieved by attaching security requirements, as metadata attributes, to vulnerable responsibilities. In addition, defense mechanisms are implemented using failure scenario paths. We have used the security tactics to build a secure development approach simpler and faster than methodologies based on threats modeling.
- A UCM describes with precision the functional behavior of an actor. However, we don't know precisely how an attacker will break the system security. If such an information is available, the vulnerabilities would have been fixed. In our approach, we specify the types of measures (using the security tactics) that the system should implement in order to detect, resist, react, and recover from an attack. Once, the threat details are available, they can be integrated within the scenario as functional behavior.

Our proposed approach relies primarily on the security tactics introduced by Bass et al. [11]. One possible threat to the validity of our approach is related to the maturity of these tactics. Indeed, a tactic is considered to be a relatively new design concept that complements the existing architectural and design patterns [4]. However, we believe that these tactics provide a comprehensive coverage of security means, that are general and flexible enough to accommodate various security requirements.

Several attempts have been proposed to revise the set of security tactics initially introduced by Bass et al. [5]. Ryoo et al. [4] have proposed a methodology for revising security tactics hierarchy through derivation, decomposition, and reclassification. However, in order to accommodate the addition of a new tactic or the refinement of an existing one, only minor changes to the UCM security-enables metamodel are required.

7 Conclusions and Future Work

In this work, we have modeled security requirements at the very early stages of the system development process, before committing to a detailed design. We have extended the Use Case Maps language with security-related features covering the well-known security tactics by Bass et al. [11]. The resulting extensions are described using a metamodel and implemented into the *jUCMNav* tool using the metadata mechanism, allowing for further model refinement and a smooth move towards more detailed design models.

As a future work, we aim at evaluating empirically our approach using real-world case studies. In addition, we plan to conduct a qualitative analysis of the efficiency of the proposed UCM-based security requirements.

Acknowledgment

The authors would like to acknowledge the support provided by the Deanship of Scientific Research at King Fahd University of Petroleum & Minerals (KFUPM) for funding this work through project No. IN131031.

References

1. ITU-T: Recommendation Z.151 (10/12), User Requirements Notation (URN) language definition, Geneva, Switzerland (2012)
2. Clements, P., Northrop, L.: Software architecture: An executive overview. Technical Report CMU/SEI-96-TR-003, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1996)
3. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice. Wiley Publishing (2009)
4. Ryoo, J., Laplante, P., Kazman, R.: Revising a security tactics hierarchy through decomposition, reclassification, and derivation. In: Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on. (June 2012) 85–91
5. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003)
6. Hassine, J.: Early availability requirements modeling using use case maps. In: Eighth International Conference on Information Technology: New Generations (ITNG), Las Vegas, Nevada, USA. (April 2011) 754–759
7. Hassine, J., Gherbi, A.: Exploring early availability requirements using use case maps. In Ober, I., Ober, I., eds.: SDL 2011: Integrating System and Software Modeling. Volume 7083 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 54–68
8. Hassine, J.: Describing and assessing availability requirements in the early stages of system development. Software & Systems Modeling (2013) 1–25
9. Hassine, J., Hamou-Lhadj, A.: Towards the generation of AMF configurations from use case maps based availability requirements. In Khendek, F., Toeroe, M., Gherbi, A., Reed, R., eds.: SDL 2013: Model-Driven Dependability Engineering. Volume 7916 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2013) 36–53
10. Hassine, J., Mussbacher, G., Braun, E., Alhaj, M.: Modeling early availability requirements using aspect-oriented use case maps. In Khendek, F., Toeroe, M., Gherbi, A., Reed, R., eds.: SDL 2013: Model-Driven Dependability Engineering. Volume 7916 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2013) 54–71
11. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. 3rd edn. Addison-Wesley Professional (2012)
12. ITU-T: E.800: Definitions of terms related to quality of service. `https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-E.800-200809-I!!PDF-E&type=items` (September 2008) [Online; accessed 15-June-2015].
13. ITU-T: X.1051: Information technology - Security techniques - Information security management guidelines for telecommunications organizations based on ISO/IEC 27002. `https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-X.1051-200802-I!!PDF-E&type=items` (February 2008) [Online; accessed 15-June-2015].

14. jUCMNav: jUCMNav Project, v6.0.0 (tool, documentation, and meta-model) (2014)
15. McDermott, J., Fox, C.: Using abuse case models for security requirements analysis. In: Proceedings of the 15th Annual Computer Security Applications Conference. ACSAC '99, Washington, DC, USA, IEEE Computer Society (1999) 55–64
16. Sindre, G., Opdahl, A.: Eliciting security requirements with misuse cases. Requirements Engineering **10**(1) (2005) 34–44
17. Firesmith, D.: Security use cases. Journal of Object Technology **2**(1) (2003) 53–64
18. Karpati, P., Sindre, G., Opdahl, A.: Visualizing cyber attacks with misuse case maps. In Wieringa, R., Persson, A., eds.: Requirements Engineering: Foundation for Software Quality. Volume 6182 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2010) 262–275