A Framework for Specifying and Managing Security Requirements in Collaborative Systems

Stephen S. Yau and Zhaoji Chen

Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287-8809, USA
{yau, zhaoji.chen}@asu.edu

Abstract. Although security has been recognized as an increasingly important and critical issue for software system development, most security requirements are poorly specified: ambiguous, misleading, inconsistent among various parts, and lacking sufficient details. In this paper, a framework for specifying unambiguous, interoperable security requirements and detecting conflict and undesirable emergent properties in collaborative systems is presented. The framework includes a core ontology representing hierarchical security requirements, an ontology-based security requirement specification process, a set of security requirement refining rules, an algorithm for automatic security requirement refinement and an analysis algorithm to detect inconsistent security requirements. In this paper, the specification and refinement of security requirements are emphasized.

Keywords - Software security, security specification, hierarchical security requirements, framework, collaborative systems, ontology, requirement refinement algorithms

1 Introduction

Software development is traditionally driven by satisfying functional requirements: what the system must do. However, non-functional requirements, such as security, performance, and interoperability, are often as important as functional requirements. Today, people increasingly rely on information systems, which often consist of software systems running on many interconnected computers with various capabilities, such as servers, desktops, laptops, cell phones and PDAs [1]. The pervasive connectivity has not only enhanced our ability to quickly exchange information and share computation resources, but also increased the chances for attackers to launch malicious attacks on information systems. With the growing concerns over system availability, data integrity and privacy, security has become an increasingly critical issue for most software systems [2, 3].

It is well recognized that requirement engineering is both important and economical to successful software development. The later the security is addressed in the development cycle, the costlier it becomes [4]. To build secure software, accurate and consistent security requirements must be specified. Despite its importance, little research has been done on integrating security in software development. Security requirements are usually generated in an ad-hoc manner, and the specified requirements may have the following undesirable properties [3]:

- *Ambiguous*: Requirements are specified in natural languages, which are usually vague and difficult to validate.
- *Misleading*: Two collaborating developers may use different terms for the same meaning or use the same term for different meanings.
- *Inconsistent*: To address different functional requirements, different developers may generate contradictory security requirements.
- *Lacking of sufficient detail information*: Some security requirements stay at very high level and lack sufficient information for them to be enforced or evaluated.

Most efforts in non-functional requirements have focused on well-specified requirements like real-time or usability, and security issues are likely left for maintenance in the infamous *penetrate and patch* manner [5]. Because various security requirements may be related, the method chosen to address a security requirement may impact other security requirements. Increasing complexity and extensibility of software, as well as dynamic operational environments, make it difficult to generate well-specified security requirements.

Although there have been several approaches to generating security requirements specification and refinement, so far no approaches can provide unambiguous semantics, automated or semi-automated refinement, and situation-awareness simultaneously. Several *product-oriented* approaches have tried to formally address non-functional requirements by quantifying them [6] and evaluating a system to what degrees it meets its requirements. However, quantitative approaches may not be suitable for addressing security requirements. For example, the possible risk of a programming bug in the core module, which can cause to a system, is much greater than that caused by the same bug in a peripheral module. A *process-oriented* approach [2] specifies security requirements as goals and refines them to specific sub-goals based on design knowledge captured. The correlation rules address how one goal may affect some other goals. This approach can adopt an agent-oriented requirements modeling paradigm to capture social relations among different entities and actors in a system as intentional dependency [7]. Both approaches cannot handle operational environment changes, where conflicting or harmonious goal interactions may change or actors may change from allies to enemies. An *aspect-oriented* approach based on UML [8] can encapsulate solutions to satisfy certain non-functional requirements as aspects, which can be integrated with a primary model based on predefined composition directives. Composed system module can be analyzed to detect any conflict or inconsistency among various aspect models and the primary model. This approach reduces the complexity of requirement engineering and facilitates fast system evolution. However, it does not address qualitative requirements and is difficult to handle large complicated systems, especially when it involves multi-party collaboration because it requires a name-match during composition. Other UML extensions [9] tried to accommodate access control in UML, but they have focused only on static design model, which is close to implementation, not security requirements.

To fully incorporate security concerns in software development, new approaches are needed to systematically address security requirement specification and management. In this paper, we present a framework to facilitate parties in large-scale collaborative systems to specify unambiguous, interoperable security requirements for software development. It can detect and remove all four undesirable emergent properties mentioned before and be used as an add-on component for other trust management frameworks [10, 11].

2 Overview of Our Approach

Our framework consists of the following five major components as shown in *Figure 1*:

C1) A core ontology for representing hierarchical security requirements.

C2) An ontology-based security requirement specification process for formally specifying security requirements based on C1).

C3) A set of security requirement refining rules and an algorithm to decompose and organize the specified security requirements into application-specific tasks according to a given operating environment.

C4) A feedback agent that records user feedbacks if the generated result is different from what the user wants, and uses this knowledge in future refining process.

C5) Inconsistency checking algorithms.

It is noted that C2 will unambiguously specify security requirements of collaborating parties based on C1 and its extensions. The refining process carried out based on C3 can establish interoperable security requirements by unifying the use of certain terminologies among multiple parties, and take care of underspecified security requirements by decomposing them into application specific subtasks with more domain knowledge and operational information. Since refinement results are automatically generated based on predefined refining rules, they may not be exactly what the users want. C4 records user feedback about the generated results and update refining rules accordingly. After the refining process, C5 will be used to detect possible contradictions or inconsistencies in requirements. If a developer wants to use a commercial off-the-shelf (*COTS*) module, the security requirements for the candidate module will be treated same as the new modules, except their implementations which are already completed.

The major innovations in our approach are the use of the core ontology and the security requirements refining rules. Our framework provides the following support:

- Semantic interoperation support: A core ontology and its extensions serve as a common base of understanding, which can help multiple collaborative parties to establish unambiguous, easy-to-understand security requirement specification.
- Requirements management sup-

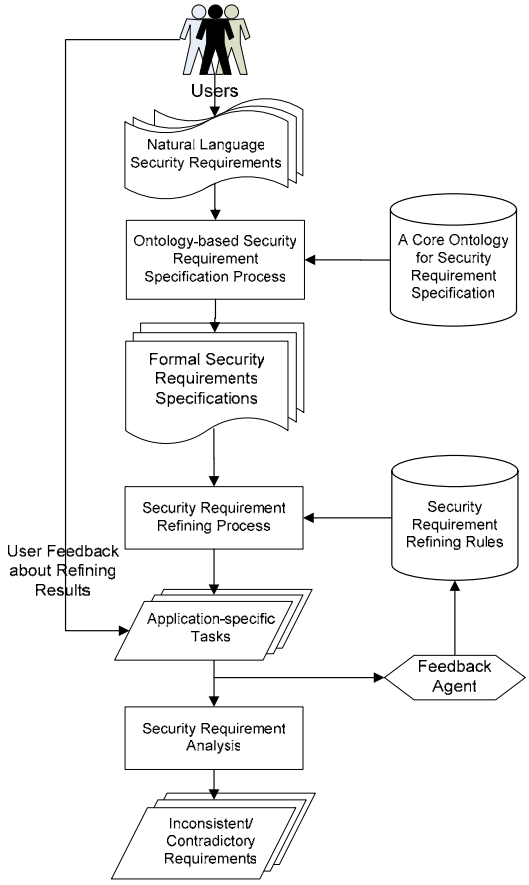

Fig. 1. Overview of our framework

port: The specified security requirements should be easy to trace, refine, reuse and prioritize.

In the following sections, we will provide detailed discussions on these components and an example to illustrate the use of our framework.

3 Security Requirement Specification

An ontology is in general a description of concepts and their relationships [12]. Because it is infeasible to list all the security concerns of possible applications in our framework, based on some common system security evaluation criteria, such as those in the "Orange Book" [13], we can only incorporate common high-level security concerns and their relations into an ontology. This ontology, which serves as a *core component* of our framework and can be extended by developers to address more application-specific security requirements, is referred as the core ontology of our framework.

The generic classes in the core ontology shown in Figure 2 are defined as follows:

- *anything* is the top of the class hierarchy for security requirement specification. It does not have any real meaning, but serves as a root class for other classes.
- *entity* is a generic security entity class which is derived from *anything*.
- *actor* is a class derived from *entity* to represent active entities within a system that carry out actions to achieve certain goals.
- *resource* is a class derived from *entity* to represent system components that need to be protected.
- *attribute* is a class derived from *anything* to describe those named values.
- *situation* is a class derived from *attribute* to identify a situation expression used in security requirement specifications. The value of a situation expression is determined by situation-aware processors [14].

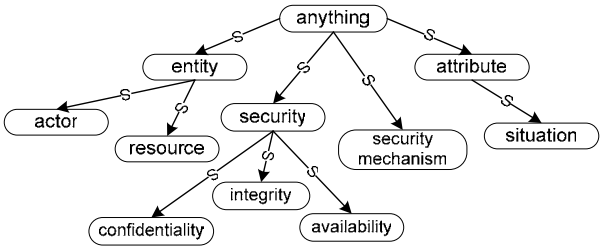

Fig. 2. The core ontology

- *securityMechanism* is a class specifies which security mechanism should be used.
- *security* is a generic class derived from *anything* to address security concerns, which can further be divided into *confidentiality*, *integrity* and *availability*.

A security requirement is essentially a goal about adding some security properties to certain parts of a system. These goals, often specified in a natural language, are usually ambiguous. This problem is exacerbated when multiple parties are involved, where different uses of the same term may create confusing statements. Based on the core ontology, a security requirement R in our framework is specified as a quadtuple:

$$R = (E, P, M, S), \tag{1}$$

where $E \in$ *entity*, $P \in$ *security feature*, $M \in$ *securityMechanism*, and $S \in$ *situation*. The quadtuple is interpreted as follows: security requirement R implies that entity E has secu-

rity property *P* by using security mechanism *M* under situation *S*. *M* can be omitted when users do not care about which security mechanism the system adopts. *S* can be omitted if property *P* applies to all situations. For example, R_1= (*account_no*, *confidentiality*, *encryption*, __) means that account numbers stored in a system should always be encrypted to ensure confidentiality. If a project is intended to use some plug-in *COTS* modules since the *COTS* modules are usually developed for some other purposes, even if they can satisfy the new functional requirements needed by the project, the *COTS* modules may not meet the security requirements of the project. Hence, the security requirements of the candidate *COTS* modules should also be specified and analyzed in the framework. Our approach to specifying security policies is summarized as follows:

RS1) Describe security requirements in a natural language.
RS2) Generate an ontology for the specified security requirements based on the core ontology as follows:
  2.1) Determine what kinds of situations we need to consider in RS1).
  2.2) Specify entities and create corresponding subclasses of `entity` in RS1).
  2.3) Specify security properties and security mechanisms in RS1), and generate subclasses for each of `security` and `securityMechanism`.
RS3) Specify requirements as quadtuples (1) using the results generated in RS2).

Because all four elements are specified based on our core ontology, each element is clearly defined. There is only one interpretation for a quadtuple, and thus a specified security requirement can only have one unambiguous meaning. However, confusing statements may still exist since different parties can derive and use their own terms based on the core ontology. Underspecified statements may also exist if a user only uses the general concepts in the core ontology, instead of extending them to express more detailed application-specific information. These two problems are taken care of by security requirement refinement which will be discussed in Section 4.

4 Security Requirements Refinement

In this section, we will discuss components C3 and C4 and the process of using them to refine security requirements. The security requirement refining rules specify how security requirements can be transformed. In our framework, there are three types of refining rules: *substitution*, *decomposition* and *combination*.

The *substitution* rules are used to provide interoperability among parties. In large collaborative systems with multiple developers involved, these developers may specify the same security requirement using different terms, or they may interpret one term with different meanings. Arranging meetings with all parties involved and specifying requirements together is one way to solve this problem. However, this approach is costly, inefficient, and sometimes infeasible. In our framework, we address this problem using the core ontology and its user-defined extensions. We assume that each team has its well-defined terminologies that all developers in the team will follow. Instead of getting all developers in all teams to work together, each team can work independently and specify its security requirements. Before the framework integrates all security requirements together, the common understanding of different teams on various terminologies can be

established by a group with a representative from each team. This common understanding is defined as *substitution* rules and our framework will automatically apply these rules during security requirement integration among different parties. A *substitution* rule specifies that if the two terms T and T' have the same meaning, and for any two security requirements R_1 and R_2, if the only difference is one uses T and the other uses T', we can substitute R_1 with R_2, vice versa. If T and T' appear to be entities, the *substitution* rule is expressed as follows:

$$R_1: (T, P, M, S) \rightarrow R_2: (T', P, M, S). \quad (2)$$

Similar rules can be specified for security properties, security mechanisms or situations.

The *decomposition* rules decompose general, high level security requirements into more specific individual tasks. Security requirements specified by various developers vary in levels of detail. Some stay at very high level, like "Accesses to the account should be controlled". Some include great details like "Access to the account can only be granted to those people listed in the access control list during normal business hours". This phenomenon sometimes is referred as *local heroes* [5], which means the specification relies heavily on individual expertise. This means that if a developer knows the area well, he is likely to put down more detailed and complete specifications. Comparing the above two requirements, the second requirement is much easier to implement, test and evaluate. But, not everyone in a development team is a local hero. Thus, from time to time, we have to deal with underspecified security requirements. In our framework, we summarize the expertise from those *local heroes* into *decomposition* rules. Based on these rules, a general, underspecified security requirement will be automatically refined by adding application-specific constraints or decomposing it into several individual sub-goals. A decomposition rule is specified as follows:

$$R_0:(E_0, P_0, M_0, S_0) \rightarrow R_1:(E_1, P_1, M_1, S_1) \wedge R_2: (E_2, P_2, M_2, S_2), \quad (3)$$

which means requirement R_0 can be satisfied by satisfying both R_1 and R_2.

The *combination* rules are used to reorganize individual tasks generated by *decomposition* rules. In a collaborative system, security requirements specified by different users may be interrelated. After applying decomposition rules, we may have several individual requirements concerning the entity with the same E**.** Although they may not be exactly the same, it is still helpful if we can combine all interrelated security requirements and address them together. When $E_1 = E_2 = E$, since R_1 and R_2 both concern with entity E, the combination of R_1 and R_2 can be carried out by combining the security mechanisms used, or by satisfying both security properties required in R_1 and R_2. If R_1 and R_2 require the same entity to use the same security mechanism under the same situation to satisfy two different security properties, the combined result should require the entity to satisfy both properties P_1 and P_2. In this case, a combination rule can be specified as follows:

$$R_1:(E_1, P_1, M_1, S_1) \wedge R_2:(E_2, P_2, M_2, S_2) \wedge (E_1 = E_2) \wedge (M_1 = M_2) \wedge (S_1 = S_2) \rightarrow R':(E_1, P', M_1, S_1), \quad P' = P_1 \wedge P_2. \quad (4)$$

Similar rules can be specified for security mechanisms or applicable situations.

For a large, distributed collaborative system, we can have hundreds of these refining rules. The number of specified security requirements for such a large system can also be very large, and hence manually applying each suitable refining rule to the specified secu-

rity requirements will not be feasible. Thus, we develop the following algorithm to automatically applying suitable refining rules:

```
Initialization:
Store all refining rules in a database table T_R,
Store each security requirement R_j in a list L_R, n is the
number of nodes, and L_R[i] is the i^th node in L_R.

Refinement process:
int count, head = 0, tail = n-1;
   while ( head ≠ tail){
    count = 0;
    for(int: i from head to tail){
      while(found L_R[i]→ R_i' in T_R)
        L_R[i] = R_i';
      if(found L_R[i]→ R_i1 ∧ R_i2 in T_R){
        delete L_R[i] from L_R;
        append R_i1 and R_i2 to the end of L_R;
        count ++;}
    }
    head = tail+1 – count;
    tail = tail + 2*count-1;
   }
  for (int:i from 0 to tail)
  for (int:j from i to tail)
    if((L_R[i]∧L_R[j]→R_k)or(L_R[j]∧L_R[i]→R_k) in T_R){
       delete L_R[i] and L_R[j] from L_R;
       append R_k to the end of L_R;
       tail --;
    }
```

After the refinement, a *feedback* agent will record user feedback on the generated results and update refining rules so that future refining results can be closer to what users expect. For example, one refining rule R_1 states that security requirement R_1: (E_1, P_1, M_1, S_1) should be decomposed into R_2 and R_3. But, there is a specific situation S_1', which is a special case in situation S_1 and has not been carefully examined in R_1. For situation S_1', users want R_1 to be decomposed to R_2' and R_3'. Based on this user feedback, a new refining rule R_2 shown in (5) specifically addressing S_1' can be added to the repository:

$$(E_1, P_1, M_1, S_1') \rightarrow (E_2, P_2, M_2, S_2') \wedge (E_3, P_3, M_3, S_3'). \quad (5)$$

5 An Example

To show how to use our framework for specifying and refining security requirements, let us consider an example of collaborative software development project for a credit card management system. This system is being developed by two teams: $Team_A$ and $Team_B$. It is desirable to use a *COTS* module developed by *CompanyC* in the project.

Team$_A$ is responsible for developing the *POS* purchase authorization module, and it has specified one security requirement: the credit card numbers must always be encrypted during communication between the *POS* machine and the central server.

Team$_B$ is responsible for developing the online account management module and it has specified two security requirements. The first requirement is that the account information must not be stored in plain text for confidentiality reason under any circumstances. The second requirement is that when a cardholder first registers, he must choose a password more than 6 characters long with at least one letter and one number.

The *COTS* module that developers want to reuse was developed to allow cardholders to access their accounts by phone. The only security requirement is that the system must ask the caller to enter his/her password before releasing any account information.

Following the three steps in Section 3, we have

RS1) Security requirements have already been described in natural languages as given above.

RS2) Generate the ontology from RS1) as follows:

2.1) Determine the situations we need to consider

- S_{A1}(pos_comm) :: *situation*: S_{A1} specifies situation when a *POS* machine is communicating with the server.
- S_{B2}(pass_requirement1) :: *situation*: password requirement for $S_{B2.}$
- S_{C1}(pass_requirement2) :: *situation*: password requirement for S_{C1}.

2.2) Specify the entities in the requirements by corresponding subclasses of the `entity` class. There are two entities in this example. Hence, we have

- *account_info*[*account_no*, *other_info*] :: *resource*.
- *creditcard_no* :: *resource*,

2.3) Specify the security features we need to address and the security mechanisms we want to adopt. In this example, we have

- *encryption* :: *securityMechanism*: the encryption class
- *authentication* :: *securityMechanism*: the authentication class
- *password* :: *authentication*: authentication can be done through password

RS3) Specify the requirements as quadtuples (1) as follows:

- R_{A1}: (*creditcard_no*, *confidentiality*, *encryption*, S_{A1})
- R_{B1}: (*account_info*, *confidentiality*, *encryption*, __)
- R_{B2}: (*account_info*, *integrity*, *password*, S_{B2})
- R_{C1}: (*account_info*, *integrity*, *password*, S_{C1})

Then, the refining process is carried out according to the algorithm in Section 4:

Initialization: Store all refining rules in T_R. First, when the system is for a credit card company, the "account number" is the same as the "credit card number". We also know that the account information is stored as the account number plus other information. Thus, in this example, we have the following rules:

$$(creditcard\_no, P, M, S) \rightarrow (account\_no, P, M, S); \quad \text{(R1)}$$

$$(account\_info, P, M, \_\_) \rightarrow (account\_no, P, M, \_\_) \wedge (other\_info, P, M, \_\_); \quad \text{(R2)}$$

$$(E, P, M, S_1) \wedge (E, P, M, S_2) \rightarrow (E, P, M, S'), \quad S' = S_1 \wedge S_2. \quad \text{(R3)}$$

Store all four security requirements specified in RS3) in a list L_R. Thus we have: n =4

and $L_R[0] = R_{A1}$, $L_R[1] = R_{B1}$, $L_R[2] = R_{B2}$, $L_R[3] = R_{C1.}$

Refinement process: Automatically apply suitable refining rules to specified security requirements. At the beginning, head=0, tail=3. We enter the while loop, go over all four security requirements in L_R,search and apply suitable substitution and decomposition rules. For the first round, $L_R[0] = R_{A1}$ matches R1, and $L_R[1] = R_{B1}$ matches R2. Thus, we have

$$R_{A1} \rightarrow R_{A1}\text{'}: (account\_no, confidentiality, encryption, S_{A1}). \quad (6)$$

$$R_{B1} \rightarrow R_{B1}\text{'}: (account\_no, confidentiality, encryption, \_) \wedge \quad R_{B1}\text{''}: (other\_info, confidentiality, encryption, \_). \quad (7)$$

We have the following operations: $L_R[0] = R_{A1}$'; delete R_{B1}; add R_{B1}' and R_{B1}''to L_R as L_R [3] and $L_R[4]$; count++, so now count =1. In the second round, the while loop only starts at the newly added requirements: R_{B1}' and R_{B1}''. But, none of them matches any *substitution* or *decomposition* rule in T_R, the while loop stops.

Now, L_R has: $L_R[0] = R_{A1}$', $L_R[1] = R_{B2}$, $L_R[2] = R_{C1}$, $L_R[3] = R_{B1}$', and $L_R[4] = R_{B1}$''. We begin to traverse L_R again to apply suitable *combination* rules. When $L_R[0] = R_{A1}$' and $L_R[3] = R_{B1}$', $L_R[0] \wedge L_R[3]$ matches (R3), we have:

$$R_{A1}\text{'} \wedge R_{B1}\text{'} \rightarrow R_1\text{:}(account\_no, confidentiality, encryption, S_1), S_1 = S_{A1} \wedge \_$$

and the following operations: delete R_{A1}' and R_{B1}'; add R_1 to L_R. When $L_R[1] = R_{B2}$ and $L_R[2] = R_{C1}$, $L_R[1] \wedge L_R[2]$ matches (R3), based on RS3) we have:

$$R_{B2} \wedge R_{C1} \rightarrow R_2\text{: } (account\_info, integrity, password, S_2), S_2 = S_{B2} \wedge S_{C1}$$

and the following operations: delete R_{B3} and R_{C1}; add R_2 to L_R.

Note that for R_1, $S_1 = S_{A1} \wedge \_ = S_{A1}$, and it is a satisfiable situation. But for R_2, S_{C1} requires users to enter password from a phone, which implies it can only be numbers. But S_{B2} requires a password to have at least one letter. Thus, $S_2 = S_{B2} \wedge S_{C1} = \varnothing$, which means this situation can never happen. This indicates there could be a conflict. No further refinement rules could be applied and the refining process is completed.

6 Conclusion and Future Work

In this paper, we have presented a framework for security requirement specification and management for collaborative systems. Our approach to specifying security requirements is easy to follow, and developers can easily adopt our approach without much effort. Our refining process is automatic. But because the complexity of this process is $O(n^2)$, where n is the number of security requirements generated in RS3), this process may take a long time to complete when n is very large. The processing time may be reduced by sorting and rearranging the security requirement list to decrease the number of iterations in the algorithm. We will complete the security requirement analysis for inconsistency detection in the framework by considering the class hierarchy among entities, security mechanisms being adopted and possible effects of satisfying each security requirement as well as under what situations these effects will occur. In addition to formalizing these correlation effects and performing system analysis, we will also focus on developing algorithms for the feedback agent.

Acknowledgment

This work is supported in part by the US Department of Defense/Office of Naval Research under the Multidisciplinary Research Program of the University Research Initiative, Contract No. N00014-04-1-0723. We would like to thank Yisheng Yao of Arizona State University for many helpful suggestions and discussions.

References

1. M. Howard and D. LeBlanc, *Writing Secure Code*, Microsoft Press, 2001.
2. L. Chung, "Dealing with Security Requirements during the Development of Information Systems", *Proc. 5th Int'l Conf. On Advanced Information System Engineering*, 1993, pp. 234-251.
3. N. R. Mead and T. Stehney, "Security Quality Requirements Engineering (SQUARE) Methodology", *Proc. Workshop on Software engineering for Secure Systems*, 2005, pp.1-7.
4. K. S. Hoo, A. W. Sudbury, and A. R. Jaquith, "Tangible ROI Through Secure Software Engineering", *Secure Business Quarterly*, vol.1(2), 2001. available at: http://www.sbq.com/sbq/rosi/sbq_rosi_software_engineering.pdf
5. J. Viega and G. McGraw, Building Secure Software: *How to Avoid Security Problems the Right Way*, Addison-Wesley, 2001.
6. S. E. Keller, L. G. Kahn and R. B. Panara, "Specifying software quality requirements with metrics", *Tutorial: System and Software Requirements Engineering, R. H. Thayer and M. Dorfman, Editors*, IEEE Computer Society Press, 1990, pp. 145-163,
7. L. Liu, E. Yu and J. Mylopoulos, "Analyzing Security Requirements as Relationships Among Strategic Actors", *E-Proc. 2nd Symp. on Requirements Engineering for Information Security (SREIS'02)*, 2002. available at: http://www.sreis.org/old/2002/finalpaper9.pdf.
8. R. France, I. Ray, G. Georg and S. Ghosh, "Aspect-oriented Approach to Early Design Modeling", *IEE Proc. Software*, vol.151 (4), 2004, pp.173-185.
9. T. Lodderstedt, D.A. Basin, and J. Doser, "SecureUML: A UML-Based Modeling Language for Model-Driven Security," *Proc. 5th Int'l Conf. On Unified Modeling Language*, 2002, pp.426-441.
10. S. S. Yau, Y. Yao, Z. Chen and L. Zhu, "An Adaptable Security Framework for Service-based Systems", *Proc. 10th IEEE Int'l Workshop on Object-oriented Real-time Dependable Systems (WORDS2005)*, 2005, pp.28-35.
11. S. S. Yau, D. Huang, H. Gong and Y. Yao, "Support for Situation-Awareness in Trustworthy Ubiquitous Computing Application Software", *Jour. Software Practice and Experience*, 2006, available at: http://www3.interscience.wiley.com/cgi-bin/fulltext/112600143/PDFSTART
12. T. R. Gruber. A translation approach to portable ontologies. *Knowledge Acquisition*, vol.5(2), 1993, pp.199-220.
13. U.S. Department of Defense, "Trusted Computer Systems Evaluation Criteria", DOD 5200.28-STD, Dec. 1985, available at: http://csrc.nist.gov/secpubs/rainbow/std001.txt
14. S. S. Yau, Y. Wang, D. Huang and H. P. In, "Situation-aware Contract Specification Language for Middleware for Ubiquitous Computing," *Proc. 9th IEEE Workshop on Future Trends of Distributed Computing Systems*, 2003, pp. 93-99.