Integrating Functional and Security Requirements Analysis using SOFL for Software Security Assurance

著者	Busalire Onesmus Emeka
出版者	法政大学大学院情報科学研究科
journal or publication title	法政大学大学院紀要. 情報科学研究科編
volume	14
page range	1-6
year	2019-03-31
URL	http://doi.org/10.15002/00021928

Integrating Functional and Security Requirements Analysis using SOFL for Software Security Assurance

Busalire Onesmus Emeka
Graduate School of Computer and Information Sciences
Hosei University
Tokyo, Japan
busalire.onesmus.39@stu.hosei.ac.jp

***Abstract*— Formal methods have been applied to define requirements for safety and/or security critical software systems in some industrial sectors, but the challenge is the lack of a systematic way to take security issues into account in specifying the functional behaviors. In this paper, we propose a formal approach to expressing and explicitly interweaving security and functional requirements. With this approach, the functional behaviors of the system are precisely specified using the Structured Object Oriented Formal Language (SOFL), the security rules are systematically explored, and the result is properly incorporated into the functional specification as constraints. The resultant specification then defines the system functionality that implies the conformance to the security rules. Such a specification can be used as a firm foundation for implementation and testing of the implementation. We discuss the principle of interweaving security rules with functional specifications and present a case study to demonstrate the feasibility of our approach**

Keywords—SOFL, Security Requirements Engineering, Formal methods, Secure by design, Attack Tree Analysis

I. INTRODUCTION

Software systems are becoming ubiquitous with software applications being used in the fields of finance, education, and transport and logistics e.t.c. With this widespread use of software applications, the security of the data handled and stored by these applications has become more and more important. This has made software security to be one of the most crucial and necessary features of high integrity software systems. However, most software engineering methodologies have a bias of taking the standard approach of analysis, design and implementation of software system without considering security, and then add security as an afterthought [1].A review of recent research in software security reveal that such approach may lead to a number of security vulnerabilities that are usually identified after system implementation. Fixing such vulnerabilities calls for a "patching" approach since the cost associated with redevelopment of the system at such a point may be too high. However, the "patching" approach is an anti-pattern in the development of high risk software systems. To solve the challenge of integrating security attributes into software system requirements, we have developed a requirement engineering methodology that promotes a systematic integration of security requirements into the software design process. Our approach pushes for: 1.) Availing to the developer a variety of security methods and their tradeoffs. 2.) Providing a systemic methodology for integrating security requirements into the software design process. This methodology advocates for a security aware software development process that combines a selected standard software development methodology, formal methods techniques, and standard security functions[2].

Our proposed methodology works by adopting the secure by design[3] software development approach through provision of a formal model for interweaving security and functional requirements. We achieve this by integrating process trees and attack trees[4] security analysis methodologies with a formal design process of functional requirement analysis and specification using Structured Object Oriented Formal Language (SOFL)[5] [6]. The process tree offers the benefit of a bounded scope, enabling the traversal of all the application`s processes from the root node to the forked child processes at the sub-nodes and end-nodes. While traversing through the nodes of the process tree, we conduct an attack tree analysis at each process node to identify potential vulnerabilities and define their mitigation strategies as additional security requirements.

The main contributions of this paper are:

- Provide a formal verifiable model for integrating security and functional software requirements.
- Mitigate security threats [7] with proper security mechanisms by formally identifying, defining and expressing potential software vulnerabilities and their related countermeasure strategies.

The rest of the paper has the following organization. Section II focuses on related existing research on secure by design security requirement engineering approaches. Section III

provides the basic concepts of our proposed methodology for interweaving functional and security requirements. To demonstrate the feasibility of our proposed approach, we present in section IV a case study through which we used the framework to generate, analyze and integrate security and functional requirements of an online banking application. Finally, section V concludes the paper by sharing our experience of the case study, lessons learnt and the future direction of our research.

II. RELATED WORKS

A number of researchers have worked on models targeting the integration of software security attributes at the requirements levels. Epstein et.al [8], proposed a framework for network enterprise utilizing UML notations [9] to describe Role Based Access Control model (RBAC). Shin et.al [10] offered a similar proposal focusing on the representation of access control such as MAC and RBAC using UML. Jurjens [11] proposed an extension of UML, called UMLsec which focusses more on multi-level security of messages in UML sequence and state interaction diagrams. Similarly, Lodderstedt et.al [12] introduce a new meta-model components and authorization constraints expressed for Role Based Access Control. These attempts leveraged on extending UML to incorporate security concerns into the functionalities provided by the software system.

Logic based approaches[14] have also been proposed for security aware requirement engineering techniques. They offer an expressive methodology for the specifications of security properties and security functions.

Mouratidis et.al [15] proposed the Secure Tropos methodology, which is based on the principle that security should be given focus from the early stages of software development process, and not retrofitted late in the design process or pursued in parallel but separately from functional requirements. However, to the best of our knowledge, the existing security requirement engineering approaches address different security concepts and take different viewpoints on matters security. Each modeling approach can express certain aspects but may lack conceptual modeling constructs to interweave security requirements with their associated functional requirements from the early stages of requirements engineering.

This paper seeks to contribute to this gap by presenting and discussing the application of a software engineering methodology, which supports the idea of secure by design approach by analyzing software vulnerabilities and incorporating recommended security considerations at the requirements engineering phase.

III. OUR PROPOSED METHODOLOGY

Our methodology for interweaving security requirements with functional requirements works by integrating functional requirements written in SOFL [6] and standard security requirements drawn from the Common Criteria for Information Technology Security Evaluation [16], the AICPA's generally accepted privacy principles and the BITS Master Security Criteria [16]. We elicit these standard security requirements using SQUARE [17] methodology. SQUARE encompasses nine steps, which generate a final deliverable of categorized and prioritized security requirements. The outcome of the SQUARE methodology is a set of standard security requirements, broadly be classified into: Identification requirements, Authentication requirements, Authorization requirements, Security auditing requirements, Confidentiality requirements, Integrity requirements, Availability requirements, Non-repudiation requirements, Immunity requirements, Survivability requirements, System maintenance security requirements and Privacy requirements. Table I below showcases a sample identification requirement for preventing backdoors in authentication systems, elicited using SQUARE methodology.

TABLE I. SAMPLE STANDARD SECURITY REQUIREMENT

Req ID: SR-IDEN-010	**Category: Security**
Subcategory(ies)/Tags	Identification, User ID, Login, Backdoor
Name	Backdoor Prevention
Requirement	All interfaces of software that are accessed for performing any action shall have the capacity to recognize the user ID
Use Case(s)	Initial login to the system, batch jobs, API calls, network interface
Rationale	Identification must be applied across all system interfaces. In the event that a "backdoor" exists through which access is granted with no identification, the security of the system would be compromised.
Priority	Critical/**High**/Medium/Low
Constraints	N/A
Comments	The term "interface" refers to the point of entry into a system. It can be a network interface, user interface, or other system interface, as appropriate
Test Case Ref #	STC-IDEN-010-1

Our key focus is to provide a framework that can holistically integrate functional and security requirements of a system software, and eventually yield software requirements that satisfy the required security requirements. The principle of integration is a basic conjunction between a functional requirement and its associated security requirement, expressed as follows.

$$S' = F \wedge S \qquad (1)$$

Where S' is the defined software requirement, F the functional requirement and S the standard security requirement related to the functional requirement. Fig 1 below highlights a conceptual schema of our proposed framework.

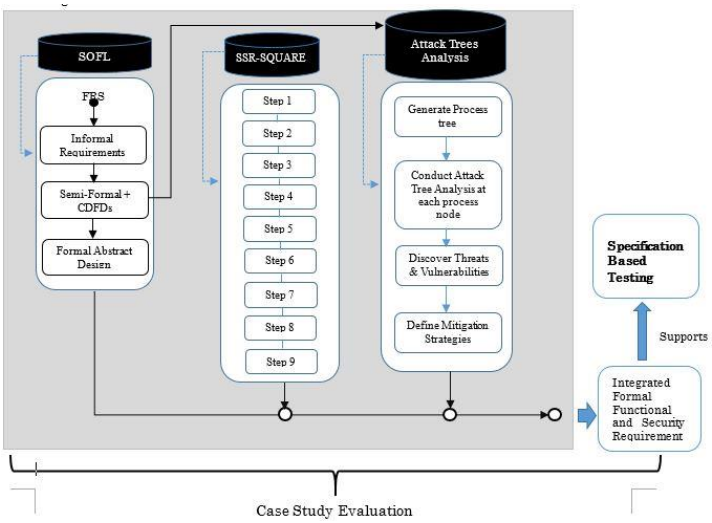

Fig. 1. Proposed framework conceptual schema

A. *The Proposed Framework in Details*

Fig 2 below shows a conceptual meta-model of our proposed framework. It illustrates the process through which we intertwine standard security requirements with functional requirements specification written in SOFL formal language given by the following steps:

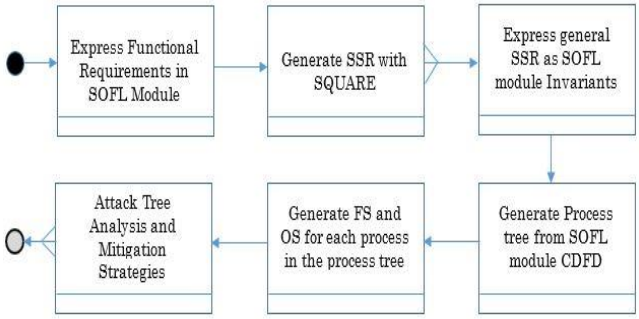

Fig. 2. A meta-model framework for interweaving security and functional requirements specifications

First, we generate the software's functional requirement specification by expressing the requirements as a SOFL module alongside its associated Conditional Dataflow Diagram. Our key goal here is to define and formerly express all the functional behaviors as a complete set of functional requirements. After defining the functional requirements, we generate relevant standard security requirements based on client specifications and application`s operating environment. We achieve this by applying the SQUARE methodology. We then express the general standard security requirements as SOFL module invariants thereby achieving the first integration of security and functional requirements.

The next step focusses on generating the application`s process tree. A process tree provides a hierarchical organization of parent processes and child processes spawned from the parent process. The generation of the process tree is achieved by converting the top level CDFD process of our SOFL module into a root process and the decomposed CDFD`s processes into child processes.

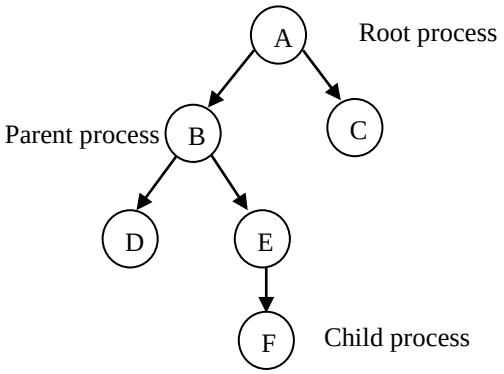

Fig. 3. An Example of a parent and child process tree

Next, we convert the parent and child processes into one or more System Functional Scenario forms. A System Functional Scenario form is a sequence of operations given by:

$$d_i[OP_1, OP_2, \ldots OP_n]d_o \qquad (2)$$

Where d_i is a set of input variables of the system behavior, d_o is the set of output variables and each $\mathbf{OP_i(i \in \{1,2,3,\ldots n\})}$ defines an operation. This System Functional Scenario defines a behavior that transforms the input data item d_i into the output data item d_o through a sequence of operations $\mathbf{OP_1, OP_2, \ldots OP_n}$. **c**ontained in a parent process or a child process at any given node of the process tree.

We then derive Operation Scenarios for the generated System Scenarios. We achieve this by transforming the pre- and postcondition of an operation into Operational Functional Scenario form consisting of operations such as

$$(OP_{pre} \wedge C_1 \wedge D_1) \vee (OP_{pre} \wedge C_2 \wedge D_2) \vee \ldots (OP_{pre} \wedge C_n \wedge D_n)$$

Where $\mathbf{C_i}$ **(i=1, … n)** is called a **guard condition** containing only input variables and $\mathbf{D_i}$ **(i=1, … n)** is known as **defining condition** containing at least one output variable.

Based on the derived Operational Functional Scenario, the final step focusses on eliciting potential vulnerabilities for each of the derived Operational Functional Scenario. We employ attack tree analysis [17] as a technique for identifying the potential vulnerabilities exhibited by each Operational Functional Scenario and further define a mitigation strategy for each of the identified potential vulnerabilities as part of the Operation Function Scenario **guard condition $\mathbf{C_n}$ or as an invariant of the SOFL module**.

These mitigation strategies qualify as additional security requirements that are intertwined with their related functional requirements.

B. *Attack Tree Analysis*

We conduct attack tree analysis through a goal oriented approach where we first identify a primary goal **X** representing a set of system assets or resources that may be targeted by an attacker i.e $\mathbf{X = \{X_1, X_2, \ldots, X_n\}}$. Our

interest here is to have an attacker`s mindset by considering what can make a functional requirement of an application fail when attacked. We then divide the primary goal into sub-goals noting that either all or some of them are required to materialize the primary goal i.e $\boldsymbol{S} \subseteq \boldsymbol{X}$. We further select a permutation $\boldsymbol{\alpha}$ of $\boldsymbol{S}$, and based on the sub tree and permutation $\boldsymbol{\alpha}$, we compute the expected outcome

Require *: The set elementary attack goal* $\boldsymbol{X = \{ X_1, X_2, \ldots, X_n \}}$ ***and*** *permutation* $\boldsymbol{\alpha \in Sn}$

1: ***for*** *i := 1 to n* ***do***
2: *Consider* $X_{\alpha(i)}$
3: ***if*** *success or failure of* $X_{\alpha(i)}$ *has no effect in the success or failure of*
4: *the root node* ***then***
5: *Skip* $X_{\alpha(i)}$
6: ***else***
7: *Try to perform* $X_{\alpha(i)}$
8: ***if*** *the root node succeeds or fails* ***then***
9: *Stop*
10: ***endif***
11: ***endif***
12: ***endfor***

Fig. 4. Attack tree analysis algorithm

IV. CASE STUDY

In this section, we demonstrate a case study from which we display a practical scenario of how our methodology achieves the interweaving functional and security requirements.

We consider a typical online banking web application where a customer can perform several financial transactions. Figures 5 showcases a decomposed CDFD of the online banking application.

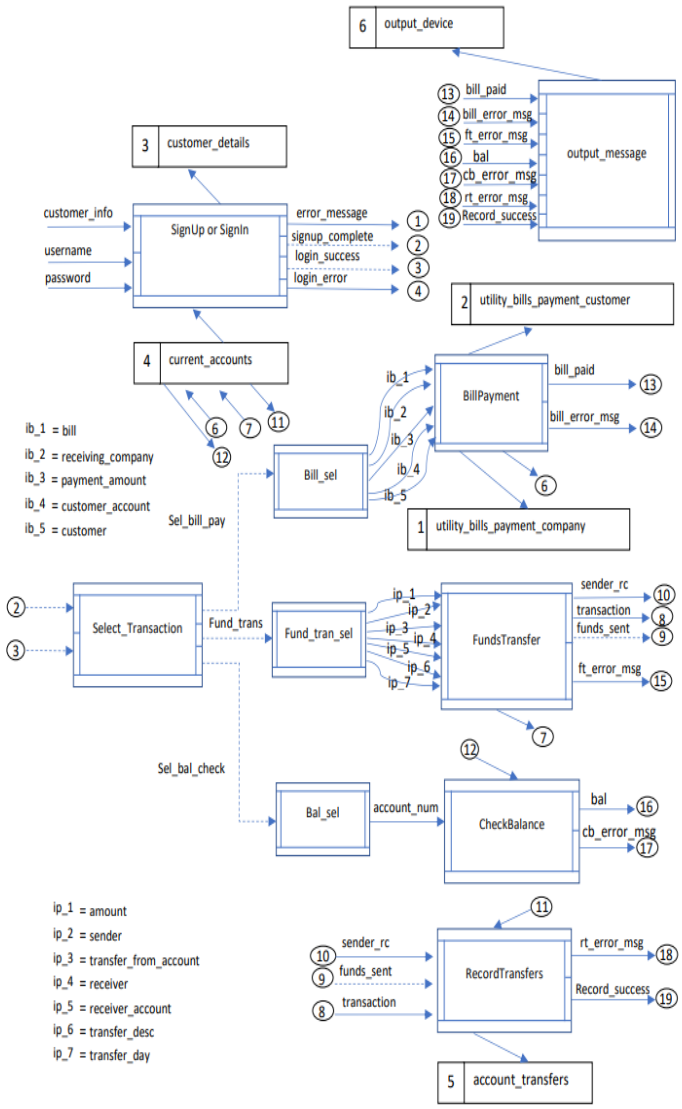

Fig. 5. Decomposed CDFD of the online banking application

From the decomposed CDFD we can generate a process tree like the one displayed below.

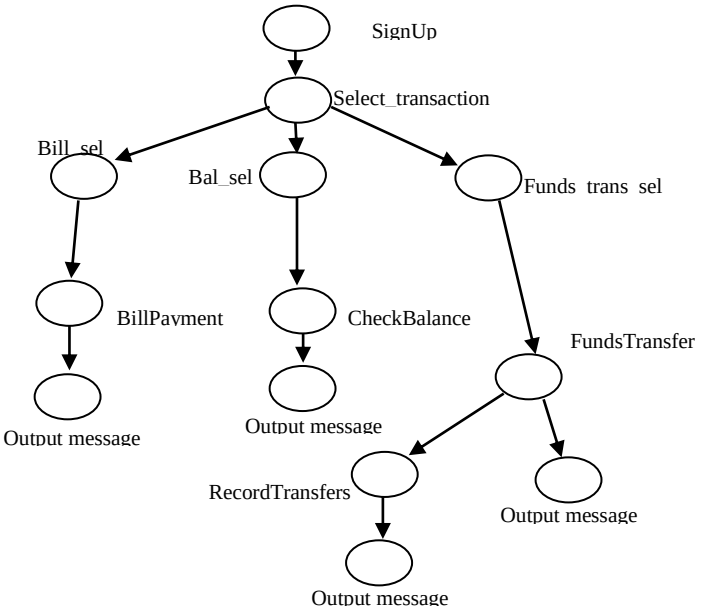

Fig. 6. Online banking Application Process Tree generated from decomposed CDFD

For the sake of brevity and clarity, we shall only highlight how we interweaved a standard authentication requirement for the *SignUp* processes whose SOFL formal specifications can be expressed as shown below.

```
/* Online banking Formal Abstract Design Specifications */

module SYSTEM_ONLINEBANKING;

/* const, var and type declarations  omitted for the sake of brevity */
inv

forall[x: CurrentAccount ] | not exists [y: CurrentAccount ] |
x.account_number = y.account_number;
/* Each customer account is unique */
forall[ x: CustomerProfile ] | not exists [y: CustomerProfile ] |
      x.email_address = y.email_address => x.national_id_num < >
y.national_id_num
/* Each customer has a unique email address and national ID number
and each customer profile is unique */

process SignUp(customer_info: CustomerProfile)
signup_complete:string
            | error_message: string

ext wr customer_details
pre customer_info notin customer_details and  not
exists[x:CustomerProfile] | x.email_address =
get(customer_details).email_address and x.national_id_num =
get(customer_details).national_id_num

Comment
 Each customer profile is unique and each customer profile has a unique
email address and national identification number

post if bound(signup_complete)
    then customer_details = union(~customer_details, {customer_info})
     and signup_complete = "Signup Successful"
    else error_message = "Either the provided email or National ID
number already exists in customer records"

end_process;
```

Fig. 7. SOFL Formal Abstract Specification for the SignUp process of the online banking application

The above SOFL specifications represent the functional behavior exhibited by our online banking application during a user sign up process. The process takes an object of customer profile information {*full_name, username, password, national_id_num, email_address}* and creates a new record of the customer if and only if the supplied customer information does not already exist in the external *#customer_details* database file where all the records of customer's profile are stored. Otherwise, it returns an error message indicating an existence of a similar record. For the login process, a user supplies a set of username and password, which are matched with those stored in the system's database.

A. *Converting the SignUp into its equivalent System Functional Scenario Form*

Given a set of customer information {*full_name, username, password, national_id_num, email_address}* as inputs, a *SignUp* process and an *error_message* as output we can generate a System Functional Scenario Form;

{full_name,username,password,national_id_num,email_address} [SignUp, ...]{error_message}

The next step involves deriving Operation Scenarios from our generated System Functional Scenario(s).

B. *Deriving Operational Scenarios*

To derive Operation Scenario(s), we take a Functional Scenario Process i.e *SignUp* and express it in the form of a chain of logical disjunction of a set of its individual conjunctive elements made up of a precondition, a guard condition containing only the input variables, and a defining condition containing at least one output variable i.e

$$(\exists_x \in current\_accounts \mid x \textbf{ not inset } current\_accounts \wedge (dom(x).full\_name = full\_name \wedge dom(x).username = username \wedge dom(x).password = password \wedge dom(x).national\_id\_num = national\_id\_num \wedge dom(x).email\_address = email\_address) \wedge signup\_complete = "Signup\ Successful")$$

OR

$$(\exists_x \in current\_accounts \mid x \textbf{ inset } current\_accounts \wedge (dom(x).full\_name = full\_name \wedge dom(x).username = username \wedge dom(x).password = password \wedge dom(x).national\_id\_num = national\_id\_num \wedge dom(x).email\_address = email\_address) \wedge error\_message = "Similar\ Records\ exist\ in\ the\ database\ already")$$

This formalized *SignUp* expression checks for the existence of similar user records before signing up a new user with the same set of record inputs. Otherwise, it returns an error message depicting the existence of a similar record with the provided set of inputs.

C. *Eliciting potential vulnerabilities for each of the derived Operational Scenarios*

Eliciting potential vulnerabilities that may be associated with our derived Operational Scenarios, involves conducting an attack tree analysis on a behavior depicted by the SignUp process node in our process tree. To figure this out, we identify a goal or resource that is part of our derived Operational Scenario and may be a subject of an attack as well as consider a standard security requirement for the same.

A typical attack tree analysis on the *SignUP* process aimed at obtaining the stored username and password or identities of their equivalent yields 4 different paths i.e direct access to the database, brute force login, threatening the user or shoulder surfing. Whereas paths such as threatening the user or shoulder surfing can be mitigated through management controls, gaining direct access to database and brute force login may not be effectively mitigated through management controls.

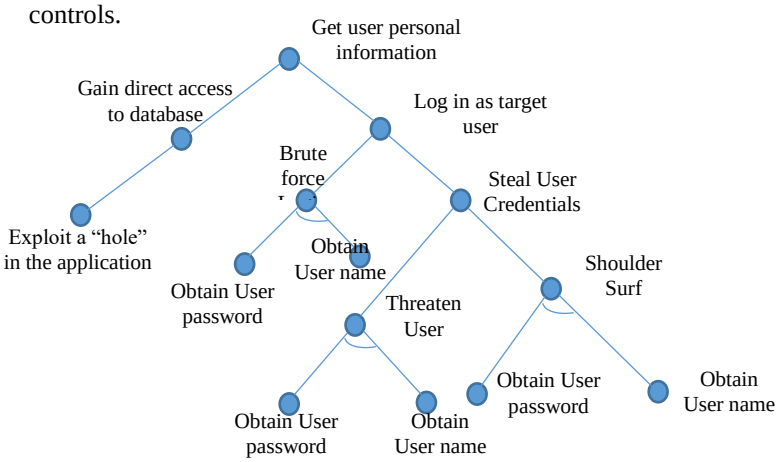

Fig. 8. Attack tree analysis showcasing paths that can be manipulated to obtain customer's personal information

To eliminate brute force login attack path, we need to interweave an authentication requirements with the functional requirements responsible for user credentials creation and storage.

Table 1 below describes a standard authentication requirement for credentials security [18]

TABLE II. CREDENTIAL SECURITY

Req. ID: SR-OBA-001	**Category: Security**
Subcategory(ies)/Tags	Authentication, Credentials, Password, Hashing
Name	Credential Security
Requirement	The system shall store the information used for authentication in a secure manner, using public and widely accepted crypto algorithms
Use Cases	Password Storage
Rationale	Authenticating information must be stored in such a way so that a third party without authorization to do so cannot easily obtain it. For example, static passwords should

Req. ID: SR-OBA-001	Category: Security
	be passed through a one-hash function and only the hash should be stored.
Priority	Critical/High/Medium/Low
Constraints	N/A
Comments	Per-user salting is recommended for storing password hashing to provide additional level of security.
Test Case Ref Number	TC-OBA-001

We achieve this by formally defining a secure way of storing the user password such as a one way hashing function given by *f(r, h(P`))* which we express as part of the *SignUP* Operarion Scenario where, *r = random number, h(P`) = Password hashing function, P` = Stored User Password*

Our strengthened Operation Scenario post-condition bearing functional and security requirements can therefore can be re-written as follows:

($\exists_x \in$ *current_accounts | x* ***not inset*** *current_accounts* $\wedge$ ***(****dom(x).full_name = full_name* $\wedge$ *dom(x).username = username* $\wedge$ *dom(x).password = password* $\wedge$ *dom(x).national_id_num = national_id_num* $\wedge$ *dom(x).email_address* **=** *email_address****)*** $\wedge$ *signup_complete = "Signup Successful"****)*** $\wedge$ *dom(x).password = f(r, h(~dom(x).password))))*

OR

($\exists_x \in$ *current_accounts | x* ***inset*** *current_accounts* $\wedge$ ***(****dom(x).full_name = full_name* $\wedge$ *dom(x).username = username* $\wedge$ *dom(x).password = password* $\wedge$ *dom(x).national_id_num = national_id_num* $\wedge$ *dom(x).email_address* **=** *email_address****)*** $\wedge$ *error_message = "Similar Records exist in the database already"****)***

V. DISCUSSIONS AND CONCLUSIONS

Our experience with the proposed framework can be summarized as follows: The methodology require some software security expertise in addition to requirement engineering skills since it focusses towards achieving the integration the integration of security analysis into the software requirement engineering process. Moreover, knowledge and skills of applying SOFL specification language in writing software requirements is a prerequisite. Even though our proposed methodology cannot guarantee the development of a completely secure system, we are confident that the application of our methodology can assist in the development of a system that is more secure compared to a system whose SRE process were done in an ad hoc manner.

This paper presents our experiences from the application of a methodology that interweaves functional and security requirements. We document our experience by applying the methodology in the development of an online banking application. Our findings indicate that the use of our approach supported the development of a software system that meets its security requirements and offers an early focus on security. Our experience on the other hand also indicated some issues for consideration, such as potential complexity of using formal notations in generating readable security requirements as well development of a supporting tool for the methodology. Resolving these issues is our main concern for future works.

REFERENCES

[1] Mouratidis, H., & Giorgini, P. (Eds.). *(2006). Integrating security and software engineering: Advances and future visions. Hershey, PA: Idea Group. doi:10.4018/978-1-59904-147-6*

[2] Khaled M Khan. *Developing and Evaluating Security-Aware Systems*, ISBN 978-1-4666-2483-2.

[3] Haralambos Mouratidis, *Integrating Security and Software Engineering: Advances and Future Visions*, ISBN 1-59904-149-9 pg 14-24.

[4] V. Nagaraju, L. Fiondella, and T. Wandji, "A survey of fault and attack tree modeling and analysis for cyber risk management," in *2017 IEEE International Symposium on Technologies for Homeland Security (HST)*, 2017, pp. 1–6.

[5] F. Nagoya, S. Liu, and K. Hamada, "Developing a Web Dictionary System Using the SOFL Three-Step Specification Approach," in *2015 5th International Conference on IT Convergence and Security (ICITCS)*, 2015, pp. 1–5.

[6] Shaoying Liu, *Formal Engineering for Industrial Software Development using SOFL method, ISBN 3-540-20602-7 Springer-Verlag Berlin Heidelberg New York.*

[7] H. C. Huang, Z. K. Zhang, H. W. Cheng, and S. W. Shieh, "Web Application Security: Threats, Countermeasures, and Pitfalls," *Computer*, vol. 50, no. 6, pp. 81–85, 2017.

[8] Epstein, P., & Sandhu, R. (1999). Towards a UML based approach to role engineering. *Proceedings of the 4th ACM Workshop on Role-based Access Control*, (pp. 75-85). ACM Press.

[9] M. P. Huget, "Agent UML notation for multiagent system design," *IEEE Internet Comput.*, vol. 8, no. 4, pp. 63–71, Jul. 2004.

[10] Shin, M., & Ahn, G. (2000). UML-based representation of role-based access control.Proceedings of the *9th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*,(pp. 195-200). IEEE Computer Society.

[11] F. Kammüller, J. C. Augusto, and S. Jones, "Security and privacy requirements engineering for human centric IoT systems using eFRIEND and Isabelle," in *2017 IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA)*, 2017, pp. 401–406.

[12] Lodderstedt, T., et al. (2002). SecureUML: A UML-based modeling language for model-driven security. *In Proceedings of UML, LNCS, Vol. 2460,* (pp. 426-441). Springer.

[13] Swamy, N., Corcoran, B., & Hicks, M. (2008). FABLE: A language for enforcing user-defined security policies. *In Proceedings of the IEEE Symposium on Security and Privacy, Oakland.*1.

[14] M. Amini and R. Jalili, "Multi-level authorisation model and framework for distributed semantic-aware environments," *IET Inf. Secur.*, vol. 4, no. 4, pp. 301–321, Dec. 2010.

[15] M. Ouedraogo, H. Mouratidis, D. Khadraoui, and E. Dubois, "An Agent-Based System to Support Assurance of Security Requirements," in *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*, 2010, pp. 78–87.

[16] "Common Criteria Part 2: Security Functional Requirements," The Common Criteria Portal, accessed March 20, 2018, https://www.commoncriteriaportal.org/cc/; "Security Criteria," BITS/Financial Services Roundtable, accessed March 22, 2018, http://www.bitsinfo.org/security-criteria/

[17] R. Kumar and M. Stoelinga, "Quantitative Security and Safety Analysis with Attack-Fault Trees," in *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*, 2017, pp. 25–32.

[18] Mark S. Merkow and Lakshmikanth Raghavan, Secure and Resilient Software, Requirements, Test cases and Testing Methods, pg 71.