Formalization of Web Security Patterns

ASHISH KUMAR DWIVEDI[1]
SANTANU KUMAR RATH[2]

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Odisha-769008, India
[1]ashish.nitcs@gmail.com
[2]skrath@nitrkl.ac.in

Abstract. Security issues in software industries become more and more challenging due to malicious attacks and as a result, it leads to exploration of various security holes in software system. In order to secure the information assets associated with any software system, organizations plan to design the system based on a number of security patterns, useful to build and test new security mechanisms. These patterns are nothing but certain design guidelines. But they have certain limitations in terms of consistency and usability. Hence, these security patterns may sometimes act as insecure. In this study, an attempt has been made to compose security patterns for the web-based application. Subsequently, a formal modeling approach for the composition of security patterns is presented. In order to maximize comprehensibility, Unified Modeling Language (UML) notations are used to represent structural and behavioral aspects of a web-based system. A formal modeling language i.e., Alloy has been taken into consideration for analyzing web-based security pattens. For the demonstration of this approach, a case study i.e., an online banking system is considered. A qualitative evaluation is performed for the identified security patterns against the critical security properties. In this study a model-driven framework is presented, which helps to automate the process of analyzing web security patterns.

Keywords: Alloy, Formal Modeling, Online Banking System, Security Patterns.

(Received March 3th, 2015 / Accepted May 4th, 2015)

1 Introduction

Developing a secure system is not an easy task as providing a protected login screen. Hence, an extra effort is required to achieve security requirements. Various security goals can be achieved by applying security patterns during design phase. Software design patterns are a set of techniques, applied during analysis and design phases for emphasizing certain design issues to recurring design problems. Each pattern emphasizes on a problem occurring in a recurring manner and provides a core solution to that problem, in such a way that one can use this solution a number of times for a particular domain [1].

In the past two decades, a number of software patterns have been identified, documented, visualized, classified, and analyzed [11] [19] [20] [26]. A good number of tools on design patterns have also been developed for detecting patterns, while instantiating of design patterns [18] [25]. These system patterns and tools facilitate the understandability and construction of systems that provide predictable, uninterrupted use of the services and resources they offer to users. Each pattern is represented using a standard pattern template that allows expressing a solution for solving recurring problems. Pattern templates are used to capture all the elements of a pattern and describe its issues, motivation, strategies, technologies, applicable scenarios, solutions, and examples. Gamma et al. [11] have proposed standard templates for their twenty three num-

ber of design patterns. Subsequently other authors have proposed different patterns, based on the template proposed by Gamma et al. [11].

Security patterns are a set of suitable techniques for analyzing, developing, and testing new security mechanisms. Security patterns document solution to resolve the design level consequences for solving recurring security problems in a particular context. Yoder and Barcalow presented seven security patterns, which are applied to various software development issues [24]. A number of other varieties of security patterns are also available in the literature [19] [20] [22].

In a pattern-oriented software development, a good number of patterns are specified using informal and semi-formal approaches, such as natural languages and other graphical notations. These notations very often lead to ambiguities and inconsistencies for the system analyst. Checking the consistency and completeness of composition of patterns using various formal methods allow detecting problems in early stages of software development [23]. Formal methods are nothing but empirical techniques for the specification, development, and verification of software and hardware systems. Formal models help to describe the software requirements precisely and unambiguously using certain tools and techniques which can capture the abstract features of a system. A number of formal specification languages are available in the literature. For analyzing security patterns a language known as Alloy has been taken into consideration, as it supports predicate logic and its syntax is helpful to express any complex constraint [9] [15]. Alloy notation can be executed by an automated tool i.e., Alloy Analyzer that performs a semantic analysis [12]. This tool performs checking of consequences, consistency, and simulated execution of the Alloy specification.

In order to demonstrate the verification of security patterns, a case study on online banking system has been taken into consideration. Nowadays, customers need more advocacy, more personal security, and more control in their banking relationships. The major challenge with different banks is that they intend to provide flexibility, shared services, easiness in use by aligning themselves to technology. These challenges are so much mingled with security issues that their solutions can be only thought by application of right security patterns. This approach is not only limited to online banking system, but it can be helpful to other systems also that perform similar types of security-based online operation. In this study, an attempt has been made to perform a qualitative analysis of software security patterns, based on a non-functional requirement i.e., security. The requirement of security is based on parameters, such as availability, confidentiality, and integrity. At the end of this study, an evaluation has been performed for the security properties and associated patterns.

The basic overview of this study is that: in the next section, related research works are presented. Third section is further divided into five subsections. In its first subsection, formal modeling framework is presented. In the second subsection, five security patterns such as *Single Sign On* [14], *Check Point* [24], *Authenticator* [4], *Policy* [4] and *Secure Proxy* [20] have been identified and it has been exhibited as to how online banking application is influenced by these patterns. In the third subsection, the structural and behavioral aspects of the composition of security patterns are presented using UML class diagram and sequence diagram respectively. In the fourth subsection, formal analysis of security requirements based on those security patterns has been performed. Subsequently, an evaluation has been performed for the identified security patterns and security properties. In the fifth subsection, model driven development of the proposed framework is highlighted. In the fifth section conclusion is presented.

2 Related Work

A good number of literatures are available for the formalization of software design patterns [17] [6] [21]. But security patterns need to formalize their notations. Some of the related works are referred as below:

Konrad et al. [16] proposed an idea on templates for security patterns. Authors considered a template of Gamma et.al. [11] and added four other fields, such as *behavior, constraints, related security patterns*, and *supported principles*. They have composed patterns of Yoder and Barcalow [24] for e-commerce application. They have proposed a technique for model checking of these set of patterns. Dong et al. [7] proposed an approach to automate verification of the compositions of security patterns by model checking. They have formally described the behavioral aspect of security patterns by using CCS (Calculus of Communicating Systems), and also proved the faithfulness of the transformation from a sequence diagram to its CCS representation.

Bayley and Zhu [3] have been proposed a metamodeling approach to the formalization of design patterns. According to them, it enables formal reasoning about patterns and their composition, transformation, and facilitates automatic tool support for applying patterns at the design stage. For the case study, authors have formally specified all Gamma et al. [11]

design patterns. Dwivedi and Rath [10] proposed an approach to incorporate security features in service oriented architecture with the help of security patterns. They have presented an architectural model integrated with security goals and security patterns. Dwivedi and Rath [8] have further analyzed a complex architectural style i.e., C2 (component and connector) using object modeling language Alloy. They have performed consistency checking for the interaction of component and C2-connector, interaction between connector and C2-connector, interaction between port and role etc.

Most of the above approaches are not based on lightweight formal modeling. Our approach is based on Alloy that is a lightweight formal modeling language. Few related works do not support graphical tool, as Alloy Analyzer generates graphical result to check the consistency of the composition of security patterns. Our approach can easily be reusable and extendable. From the above mentioned analysis, it may be observed that the application of security pattern is a research topic and need exhaustive thrust in order to make the design robust.

3 Proposed Work

3.1 Modeling Framework for Web Security Patterns

In this study, a modeling framework is presented for the analysis of web security patterns, which is shown in Figure 1. Model driven architecture supports an automated transformation of one model into another. The transformation process can be specified by using a number of axioms, which support mapping of source metamodel to target metamodel. The UML diagrams are very often used to specify software patterns, which need to be formalized. A tool i.e., UML2Alloy can be considered for the purpose, which helps in automatically transforming UML class digram to Alloy specification [5] [2]. In this study, a model driven framework has been proposed for modeling of web security patterns, which help to map web security problems to security patterns and then transform to a formal model i.e., Alloy specification. This Alloy model can be verified using a tool i.e., Alloy Analyzer. If the proposed model satisfies all predefined assertions then it helps in mapping the assertions to code.

3.2 Identification and Composition of Security Patterns

A good number of security patterns have been proposed to preserve the security properties such as authentication, integrity, non-repudiation, confidentiality,

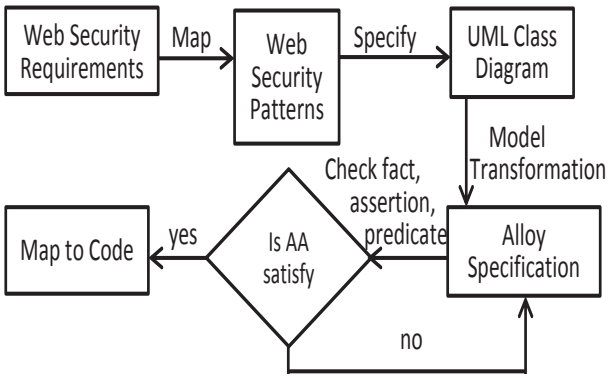

Figure 1: Model driven framework for the Web Security Patterns

availability, and authorization. Related to these security properties, a number of security threats such as spoofing, tampering, repudiation, information disclosure, denial of service, elevation of privilege are available which can affect the system. These security threats are dangerous for financial transactions, such as fund transfer in online banking system, online bill payment, loan application, and other online transactions. Hafiz et al. [13] described a pattern language for security aspects, having *ninety six* patterns. Although for any application, all patterns may not be applied; but based on security requirements for the case study on online banking system, five software security patterns such as *Single Sign On*, *Check Point*, *Authenticator*, *Policy*, and *Secure Proxy* have been considered. Composition of identified patterns is a difficult task in design patterns. Composition of identified security patterns along with external environment, rules, and resources is presented in Figure 2.

Figure 2 represents the composition of five security patterns along with *User*, *Resources*, and *Rules*. Nowadays, each application needs a login procedure to access the system. There is a context, when an user wants to access a service through the Internet; it requires a *Single Sign On* (SSO) pattern. This pattern is necessary for consideration, otherwise the user may be forced to authenticate prior to every web service call or cache the client's credentials within the application. *Single Sign On* pattern solves this problem by entering user name, password, and some other configuration setting which are required. A *Singleton* pattern can also be used for the login class, when only one user logs into the system.

Single Sign On is used to ensure that *Check Point* gets initialized. Generally, *Check Point* can be implemented by combining other patterns such as *Strategy* and *Observer*. When a request arrives, the check point

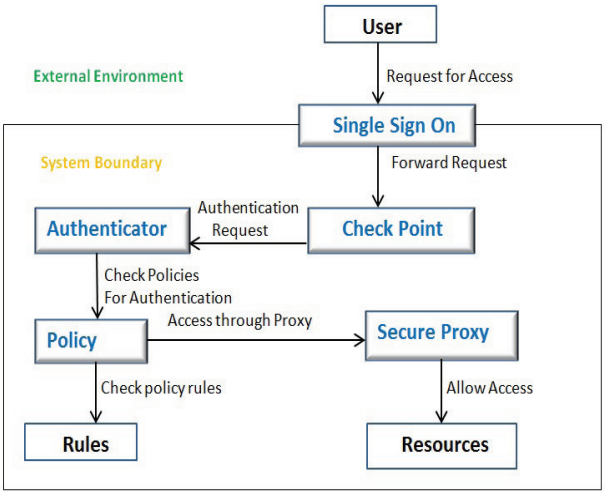

Figure 2: Composition of identified patterns

consults with the *Authenticator* to determine whether the access is permissible or not. *Authenticator* authenticates user by using *Policy* that takes decision to grant or deny access to resources based on the user's attributes, rules, policies, and other security constraints. If policy rules are not violated, user can access resource by using *Secure Proxy*.

The design of *Check Point* depends on any system's security policy. According to the policy of the system, the design might change as and when required. Hence, the algorithm for *Check Point* should be very carefully designed. *Authentication* can be performed by adopting two policies, such as direct authentication and brokered authentication. Authentication policy is based on two elements such as identification and credential. Sometimes, during the process of authenticating, the user can have a negative impact on performance due to authentication process. *Authenticator* pattern is also known as *Pluggable Authentication Modules*. *Policy* pattern is known as *Access Decision Function* as well as *Policy Decision Point*. *Secure Proxy* pattern is based on *Policy* and associated with *Rules*. In terms of security context, *Rules* are also called as guards. *Secure Proxy* is alternatively known as *Defense In Depth*, *Delegation*, *Nested Protected System* etc. It can be implemented with other patterns such as Facade.

These selected security patterns have both advantages and disadvantages. *Check Point* transfers system-state to the safe-state when system fails. But, it diagnoses faults when it is executed. Its implementation may not be that much hard. The advantage of using *Single Sign On* pattern is that it is easily implemented. But it is less secure. The *Authenticator* pattern is used almost in all cases where security concept required. The reliability and security of *Authenticator* are not much higher. *Policy* is based on security rules. If rules fulfill all security requirements for an application, it will be robust against threats. The reliability and security of this pattern depend on security strategies and rules. But it is very hard to implement. *Secure Proxy* has high speed of operation. If security policy is not satisfied, it cancels the resource request.

3.3 Structural and Behavioral aspects of patterns

In general, each pattern such as architectural pattern, security pattern, analysis pattern etc. has four essential elements, such as *context*, *problem*, *forces*, and *solution*. In this approach, *context* of identified patterns can be defined as; when a user wants to access a resource from the Internet, it requires the user to present essential information. So that, he can be accepted as an authorized user for that resource. In this context, the *problem* is that, how does the system verifies the user's information. A number of forces for the identified patterns may be taken into consideration. First, the information presented by the user to the system may be based on shared secrets, i.e., password. Second, it may be possible that the access of the resource is so simple that it does not require *Single Sign On* pattern. Third, the user and online services are supported to trust one another so as to manage security policy rules. One of the solutions of the problem is that, the user's information should be verified on the basis of security rules. If the user fulfills all the security requirements for the particular resource, he may be allowed to access that resource. Apart from these pattern's elements, there are other elements which have the impact, such as motivation, applicability, participants, collaborations, consequences, implementation, related patterns etc. These elements are called as pattern templates. Each pattern is defined in terms of its requisite templates.

For the structural demonstration of the case study, class diagram based on UML notation of online banking system is presented in Figure 3. In this diagram, all identified patterns are considered as classes. The *Bank* class is related to *BankDataBase* and *BankServer*, using composition relationship. *Customer* class is associated with *BankServer* class through *SecureProxy* class. *Customer* class is associated with *SingleSignOn* class to access different online banking services such as *Fundtransfer*, *ChequeService*, *BillPayment*, *Utility*, and *ViewTransaction*. *SingleSignOn* takes decision to allow or deny access based on user attributes, target attributes, policy rules etc. The diagram has an *Account* class which is associated with *Bank* class. *Policy* class

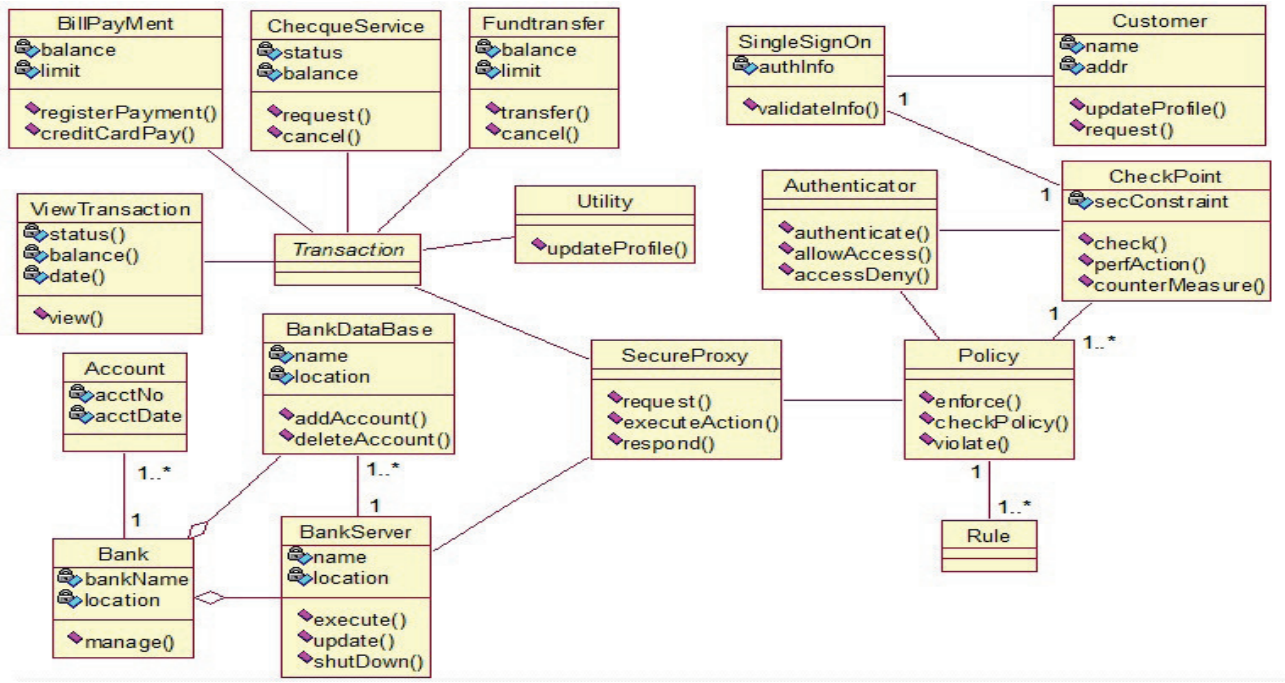

Figure 3: Class diagram of the composition of Security Patterns

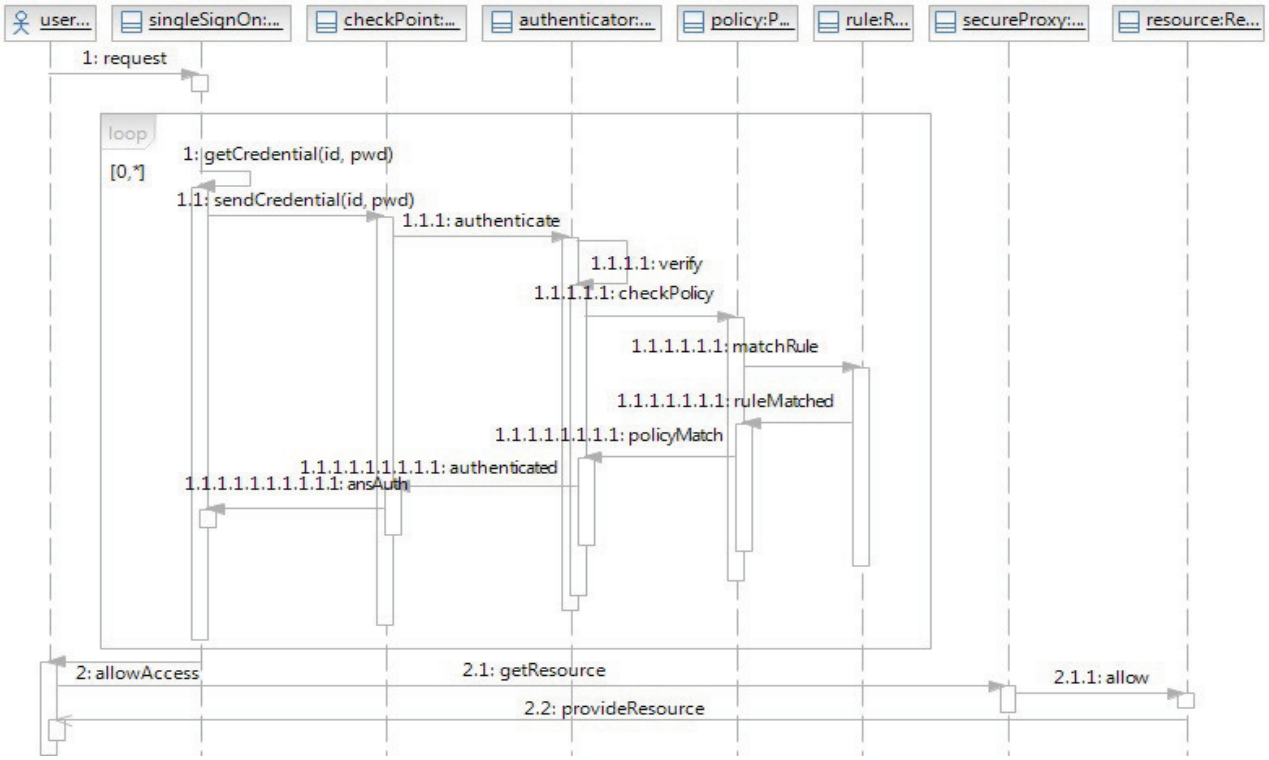

Figure 4: Sequence diagram of the composition of Security Patterns

is associated with *CheckPoint*, *SecureProxy*, and *Rule* class. Good number of components of a system are required to enforce the *Policy*. The enforcement policy invokes *enforce* method when an access is attempted to a resource. Hence, *Policy* class has a method known as *enforce*.

Figure 4 shows the behavioral aspect of composition of selected security patterns. It presents a typical scenario when a user successfully plans to login a system. *Single Sign On* could be the first step for the login procedure. *Single Sign On* initializes the system by using *Check Point*. It checks for the authentication of user. *Authenticator* checks policies. Each *Policy* is defined by some rules. If user satisfies these rules, he will be allowed for access. Finally, user gets access to resource through a *SecureProxy*.

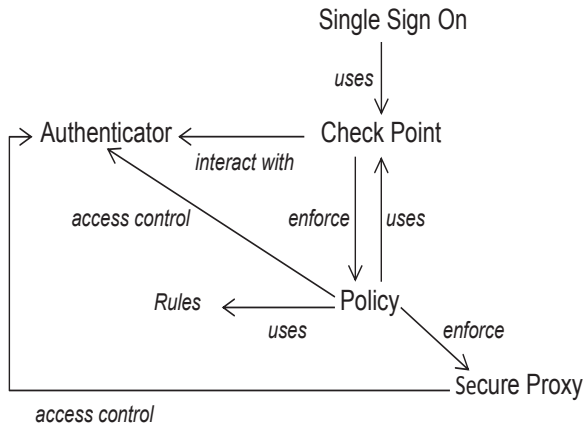

Figure 5: Pattern language for the selected Security Pattens

The pattern language proposed by Alexander was having *253 patterns* [1]. In order to describe the flow of usage of the patterns selected for online banking system, a particular language is used as shown in Figure 5. The pattern language depicts a small portion of security patterns, which applies to a software system, that need to be considered with its functional domain. This pattern language describes, as to how all the patterns can be fitted together. According to this language, selected patterns collaborate as an application security module. When a user wants to access a system, he has to enter through single entry point i.e., Single Sign ON (SSO). SSO uses *Check Point*, which interacts with *Authenticator* to validate the user's information. After validating user's information, *Check Point* enforces *Policy* to check the policy rules for accessing particular resource. *Policy* uses rules to check the limitations of user access and enforces *Secure Proxy* to provide secure access. Very often, *Check Point* can be used by other applications, when they need to perform secure transaction that cannot be performed at startup. Security checks can also be accomplished by using access control. *Policy* and *Secure Proxy* call *Authenticator* to perform access control, if necessary.

3.4 Formal specification of patterns using Alloy

Analyzing system requirements using formal modeling language Alloy has a number of advantages. Firstly, it provides an executable notation, that ensures about the functionality of Alloy model, having an unambiguous and consistent semantics. Secondly, the Alloy Analyzer translates high-level, declarative, relational expressions of the Alloy notation into a SAT (Satisfiability) formula. These SAT expressions can be executed by using a number of SAT solvers such as, *SAT4J*, *Zchaff*, *MiniSAT* etc. Thirdly, it can visualize an Alloy notation of unbounded size and subsequently specifies it to a bounded size. In order to make the pattern notations more precise, Alloy is chosen as a correct methodology for analyzing essential requirements of security design patterns. Behavioral properties of these patterns can be represented in the form of predicate logic, which can be verified by using Alloy Analyzer tool.

```
module Security_Patterns
open util/ordering [SecurityPatterns] as SP
enum AccessScheme {allow, deny}
enum Rules {Valid, Invalid}
sig Account {}
abstract sig State {}
one sig WaitForAuth, Idle, Authenticated, Busy,
    Reset extends State {}
abstract sig Operation {}
sig Fundtransfer, CheckService, BillPayment,
Request, CheckAuth, Login, RESET
                extends Operation {}
sig SecurityPatterns { state : oneState, op :
                              Operation }
```

The Alloy specification of software security patterns has a module of Security_Pattern to divide a model into other submodules. An Alloy *module* supports constraints that can be reused in different situations. In this specification, two enumerations, such as *AccessScheme* and *Rules* are considered. Alloy *enumeration* can also contain a number of atoms. During the simulation process, Alloy Analyzer considers all instances for the given problem size. Hence, the number of atoms become much large; as a result an explicit enumeration ap-

pred *reqSSO* [*sp*, *sp′* : *SecurityPatterns*] { *sp.state* = *Idle* && *sp′.op* = *Login* &&
sp′.state = *WaitForAuth* }
pred *checkAuth* [*sp*, *sp′* : *SecurityPatterns*] { *sp.state* = *WaitForAuth* && *sp′.op* = *CheckAuth*
&& *sp′.state* = *Authenticated* }
pred *reqResource* [*sp*, *sp′* : *SecurityPatterns*] { *sp.state* = *Authenticated* && *sp′.op* = *Request*
&& *sp′.state* = *Idle* }
pred *useResource* [*sp*, *sp′* : *SecurityPatterns*]
{ *sp.state* = *Idle* && *sp′.op* = (*Fundtransfer* + *CheckService* + *BillPayment*)
&& *sp′.state* = *Busy* }
pred *reset* [*sp*, *sp′* : *SecurityPatterns*] { *sp.state* = *Idle* && *sp′.state* = *Reset* }
pred *init* [*sp*, *sp′* : *SecurityPatterns*] { *sp.state* = *Idle* }
pred *traces* { *init*[*SP* / *first* []] && **all** *sp* : *SecurityPatterns* − *SP* / *last* [] |
let *sp′* = *SP* / *next* [*sp*] | ((*reqSSO* [*sp*, *sp′*] && *sp′.op* = *Login*) **or**
(*checkAuth* [*sp*, *sp′*] && *sp′.op* = *CheckAuth*) **or** (*reqResource* [*sp*, *sp′*] && *sp′.op* = *Request*) **or**
(*useResource* [*sp*, *sp′*] && *sp′.op* = (*Fundtransfer* + *CheckService* + *BillPayment*))
or (*reset*[*sp*, *sp′*] && *sp′.op* = *RESET*))}
pred *perfTrans* { *traces* && (*SP* / *last* []).*op* = *RESET*
&& *RESET* ! **in** (*SecurityPatterns* − *SP* / *last* []).*op* }

pear to be infeasible. Pruning techniques (analysis technique) are used by Alloy Analyzer tool to overcome this type of problem. These techniques rule out whole sets of atoms at once. In this model, large number of signatures have been taken into consideration, such as *State*, *Operation*, *SecurityPatterns*, *USER*, *SingleSignOn*, *Authenticator*, *CheckPoint*, *Policy*, and *SecureProxy*. The *State* and *Operation* signatures are an abstract type. For the *State* signature, many other concrete states such as *WaitForAuth*, *Idle*, *Authenticated*, *Busy*, and *Reset* have been considered. Similarly, other concrete operations such as *Fundtransfer*, *CheckService*, *BillPayment*, *Request*, *CheckAuth*, *Login*, and *RESET* have also been considered. The signature *SecurityPatterns* contains two fields, such as *state* and *op*.

3.4.1 Analyzing behavioral aspects of Security Patterns

In the process of model checking, analysis can expose defects that software architect may not have traced until much later. Alloy specification supports a logical constraint known as *fact*. In the process of rigorous analysis a fact should always hold. Alloy supports *fact, assertion,* and *predicate* for consistency checking of a system requirement. If a designer/developer wants to check the Alloy model with other axioms, and also to analyze whether these axioms are related to some other axioms or not, *predicate* is used to achieve all these requirements. A predicate is a logical formula with declaration parameters. It describes a set of states and transitions, by using constraints among the Alloy atoms and their fields. Without using a predicate, graphical results may not be generated for the operations, except from assertion's counterexamples.

The goal of this subsection is to analyze dynamic behavior of composition of selected security patterns. In this approach, an attempt has been made to capture the evolving states of the security patterns. Here, different operations are specified using predicates. First predicate *reqSSO* presents the user, who has just requested for single sign on process. In this specification, states of a system are presented in the form of pre-state and post-state, which can be expressed as *before (sp)* and *after (sp')*. Here *sp* is an instance of *SecurityPatterns*. Similarly, another operation is *checkAuth*, which represents the states of a system during the authentication process. If the user is successfully authenticated, he requests for the required resource. During this process, system's state might change, which is presented by third predicate. If user's request is accepted by the system, user can use his resource which is represented by the fourth predicate. In order to check the proper sequence of these operations, predicate *traces* are used. This predicate demonstrates the valid traces in the system. To further accomplish this process, two other predicates such as *init* and *reset* are also being used.

The predicate *traces* indicates that the initial condition holds for the initial time step, and then for all subsequent times, system need to change in accordance with one of the five predicates such as *reqSSO*, *checkAuth*, *reqResource*, *useResource*, and *reset*. The operations performed by the user is also presented here in order to help in annotation. Five operations performed by the user, which are: login to access a required resource (*Login*), check for authenticity of user (*CheckAuth*), if user is authenticated then request for resource (*Request*), if resource is available, it is provided to user (*Fundtransfer or CheckService or BillPayment*), and after using the system's service, it makes a logout (*RESET*). Along with the traces of operations, post completion error checking is also an essential task. The post completion error can be checked by using a sequence of operations, which represent a single interaction of the user with the system. Predicate *perfTrans* starts with initial state (waiting for login) and finishes with the system's logout.

3.4.2 Modeling of individual Security Patterns

Each security pattern is specified in terms of signatures, facts, predicates, and assertions. The signature *USER* has a field *sso*, which is an instance of *SingleSignOn*. For a *SingleSignOn* to be associated with a system, its user field must point to itself and it needs to have a *CheckPoint*. The signature *SingleSignOn* has two fields such as *has* and *user*. Another signature is *CheckPoint* which has also two fields such as *accessType* and *checkk*. The field *checkk* represents *Authenticator* which maps to *Policy*. In this signature, for all authenticator and policy, user is authenticated by using *Authenticator*, where rules are enforced by using *Policy*, and access type should allow the valid rules.

The signature *Authenticator* has three fields such as *accessType*, *auth*, and *cp*, which map to *AccessScheme*, *USER*, and *CheckPoint* respectively. *Policy* signature also has three fields such as *accessType*, *enforce*, and *cp* which map to *AccessScheme*, *Rules*, and *CheckPoint* respectively. The last pattern *SecureProxy* has two fields such as *accessType* and *make*. The field *make* maps *Account* to *Operation*. All these security patterns extend signature *SecurityPatterns*.

3.4.3 Formal verification of Security Patterns

For the process of formal verification, the chosen Alloy model supports a good number of templates such as *fact*, *predicate*, and *assertion*. In order to check the consistency of security provision, the following codes indicate two predicates such as, *supported* and *authPolicy*. The first predicate *supported* ensures that on a particular account user can perform operation. The second predicate *authPolicy* specifies that for an specified policy, authentication should be checked.

---

pred *supported* [*acct* : *Account*, *op* : *Operation*]
{ *acct* − > *op* **in** *SecureProxy.make* }
pred *authPolicy* [*p* : *Policy*, *a* : *Authenticator*]
{ *a* − > *p* **in** *CheckPoint.checkk* }

In this formal model, predicate *authenticated* specifies that to perform an operation on a particular account, the user should be authenticated through the policy rules. Whereas, the predicate *unauthenticated* specifies that, if policy rules are not satisfied, the user will not be authenticated.

---

pred *authenticated* [*acct* : *Account*, *user* : *USER*,
op : *Operation*]
{ **all** *p* : *Policy* | **all** *a* : *Authenticator* |
p − > *a* **in** *CheckPoint.checkk* &&
supported [*acct*, *op*] &&
user = *Authenticator.auth* && *authPolicy* [*p*, *a*] }
pred *unauthenticated* [*acct* : *Account*, *user* :
USER, *op* : *Operation*]
{ **all** *p* : *Policy* | **all** *a* : *Authenticator* |
authPolicy [*p*, *a*] && *supported* [*acct*, *op*] &&
user ! = *Authenticator.auth* }

The predicate *performed* indicates that, if predicates *supported* and *authenticated* are satisfied, then user can perform operation. Whereas, predicate *noPerformed* indicates that if predicates *supported* and *unauthenticated* are satisfied then user can't perform operation on a particular account.

In this study, a *fact* expression is specified for validating user's authenticity. According to specified fact, if the user starts an operation on an account, they must be authorized to do so. This fact statement should be valid through the specification. If any constraint in the model is violated, the fact statement generates an error message.

An *assertion* is a constraint, which can also be used for generating instances. If assertion does not hold, it generates counterexample. The assertion *Operation_Performed* is used to specify the different operations, such as transfer fund, check service, bill payment etc. for the example of online banking system. This assertion generates counterexamples, which are helpful to explain the intricacies properties.

sig *USER* { *sso* : *SingleSignOn* } { **all** *s* : *SingleSignOn* | *s* **in** *sso* **implies**
s.user = **this** && *s.has* = *CheckPoint* && *s.user* = *USER* }
sig *SingleSignOn* **extends** *SecurityPatterns* { *has* : **one** *CheckPoint*, *user* : *USER* }
sig *CheckPoint* **extends** *SecurityPatterns* { *accessType* : *AccessScheme*,
checkk : *Authenticator* −> *Policy* }
{ **all** *a* : *Authenticator* | **all** *p* : *Policy* | *a.auth* = *USER* && *p.enforce* = *Rules* &&
(*a.cp.accessType* = *allow* **implies** *p.enforce* = *valid*) }
sig *Authenticator* **extends** *SecurityPatterns* { *accessType* : *AccessScheme*,
auth : **set** *User*, *cp* : *CheckPoint* }
sig *Policy* **extends** *SecurityPatterns* { *accessType* : *AccessScheme*, *enforce* : **set** *Rules*, *cp* : *CheckPoint* }
sig *SecureProxy* **extends** *SecurityPatterns* { *accessType* : *AccessScheme*, *make* : *Account* −> *Operation* }

Modeling of individual Security Patterns

pred *performed* [*acct* : *Account*, *user* : *USER*,
op : *Operation*]
{ *supported* [*acct*, *op*] &&
authenticated[*acct*, *user*, *op*] }
pred *noPerformed* [*acct* : *Account*, *user* : *USER*,
op : *Operation*]
{ *supported* [*acct*, *op*] &&
unauthenticated[*acct*, *user*, *op*] }

fact { **all** *acct* : *Account* |
all *op* : *Operation* | **all** *user* : *USER* |
support[*acct*, *op*] && *authenticated*[*acct*, *user*, *op*]
⇒ *performed*[*acct*, *user*, *op*] }

3.4.4 Model checking using Alloy Analyzer

Alloy Analyzer is used for generating and visualizing instances and their relationship. In this formal model, operation *perfTrans* is executed for the scope value three. The graphical representation of this operation is shown in Figure 6, where it shows different instances, such as enumerations, signatures, and connections between these signatures. The graphical result shows the different types of relationships, such as Authenticator has a state i.e., *busy*, Secure Proxy has an accessType i.e., *allow*, check Point has an operation i.e., *Request*, etc. As researcher says, if the number of instances are greater than seven, Alloy Analyzer can generate all possible types of relations among given instances [15].

assert *Operation_Performed*
{ **all** *a* : *Authenticator* | **all** *sso* : *SingleSignOn* |
all *user* : *USER* | **all** *p* : *Policy* |
all *sp* : *SecureProxy* | *a.auth* == *user* &&
p.enforce = *valid* && *sp.op* =
(*Fundtransfer* + *CheckService* + *BillPayment*)
&& *sso.has.accessType* = *allow* &&
supported[*acct*, *op*] ⇒ *performed*[*acct*, *user*, *op*] }

3.4.5 Experimental results generated by Alloy Analyzer

Alloy Analyzer also helps for the generation of test cases. It does not generate test cases directly. By using Alloy Analyzer, user can find large number of relationships among the instances, which are helpful for checking the consistency of a particular method. In Table 1, first column represents the problem size (number of instances), second column indicates simulation time taken by Alloy Analyzer, third column is for the number of test cases, and fourth column represents the number of variables generated by Alloy Analyzer.

Table 1: Test cases generated by Alloy Analyzer

# Instances	Time in ms	# Test cases	# Variables
2.	161	3	1561
4.	230	17	9880
6.	600	235	58495
8.	3065	1614	95026
10.	9826	14062	2138052

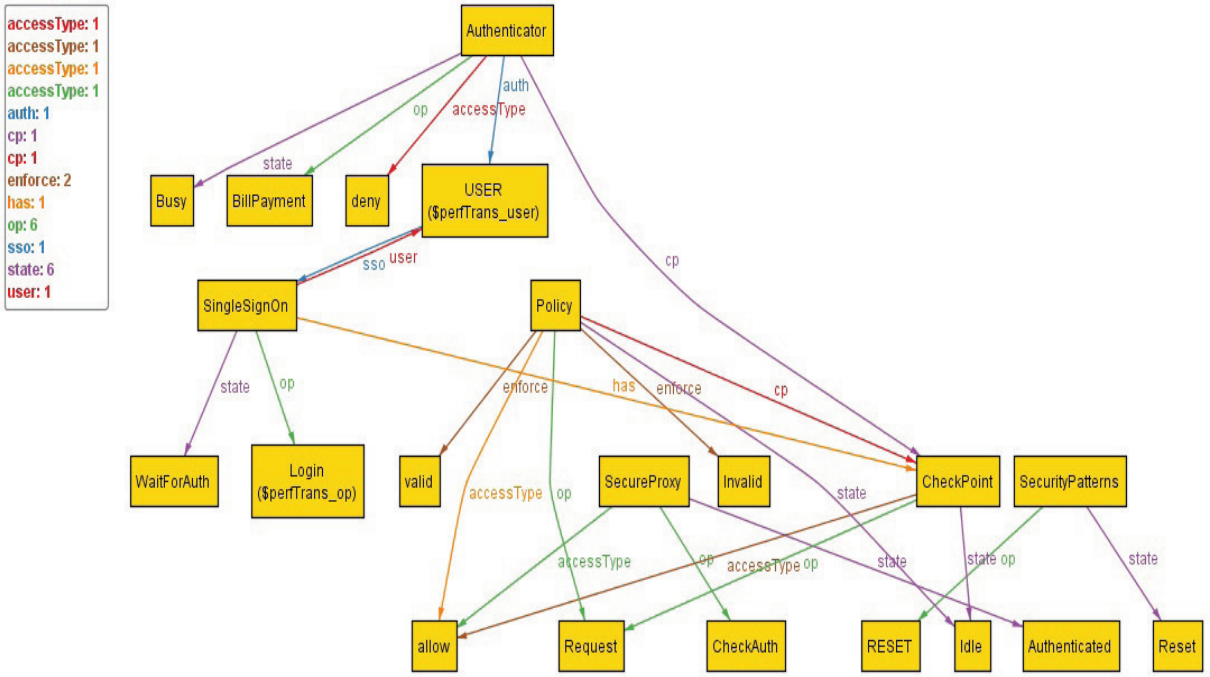

Figure 6: Graphical result generated by Alloy Analyzer

Table 2: Evaluation of Security Patterns

S. No.	Properties	Threats	Patterns
1.	Confidentiality	Information disclosure	Check Point
2.	Integrity	Tampering	*Authenticator*, *Policy*, *Secure Proxy*
3.	Availability	Denial of Service	SSO
4.	Authentication	Spoofing	Authenticator
5.	Authorization	Elevation of Privilege	Authenticator
6.	Non-repudiation	Repudiation	Secure Proxy

3.4.6 Evaluation of Security Patterns with security properties

The evaluation of these security properties, such as *confidentiality*, *integrity*, *availability*, *authentication*, *authorization*, and *non-repudiation* against the identified patterns, such as *Single Sign On* (SSO), *Check Point*, *Authenticator*, *Policy*, *Secure Proxy* are presented in Table 2. *SSO* provides a single login screen to all external entities of the system, which helps the system to trace the unusual requests by maintaining the *availability* of the system for other entities. *Check Point* ensures the *confidentiality* of system by authenticating the user and it also enforces certain security policies. Even it tries to penalize the user for violating security policies. *Authenticator*, *Policy*, and *Secure Proxy* maintain the integrity of a system. *Authenticator* is helpful to preserve the *Authentication* and *Authorization* properties. *Non-repudiation* can be maintained with the help of *Secure Proxy*.

3.5 Mapping Web Security Patterns into Formal notation: An MDA based Approach

A model driven architecture (MDA) uses the concept of metamodel that represents its elements and their relationships. The MDA approach is mainly used to perform model transformation. UML class diagram of the composition of web security patterns needs to be transformed into Alloy notation for making an automated analysis. UML class diagram can be formalized using OCL (Object Constraint Language) notation. But

OCL is not a lightweight notation and it can not generate graphical result. Hence, a UML class diagram incorporated with OCL expression needs to be transformed into Alloy expression. Table 3 represents the correspondence between UML elements and Alloy elements. The translation rules for the modeling framework are described as follows:

- Web security requirements into security patterns need to be represented.
- Security patterns using UML class diagram need composition.
- UML class diagram may be transformed into Alloy notation by using UML2Alloy.
- The axioms (fact, assertion, and predicate) using Alloy Analyzer may be checked.
- If predicate becomes inconsistent, Alloy specification need to be changed.

Table 3: Correspondence between Alloy elements and UML elements

S. No.	UML Elements	Alloy Element
1.	Package	Module
2.	Class	Signature
3.	Attributes	Relation
4.	Association	Relations
5.	OCL Expr	Assertion & Fact Expr
6.	Void Operation	Predicate
7.	Return Operation	Function
8.	Multiplicity	Cardinality

In this study web security requirements are verified using assertion, fact, and predicates that provide a security verification framework for the other similar types of system. The behavioral aspect of security patterns are also analyzed that support run time verification of a system.

4 Conclusion

Security design patterns reuse effective software design experience on solving critical security related problems. Patterns, with good error detection and correction ability, lower data redundancy, and easy implementation are useful for the system. In this study, an attempt has been made to systematically present a modeling approach to the formalization of security design patterns. This approach captures both structural and behavioral aspects that specify variants using formal modeling language Alloy. It consists of modeling of the composition of security patterns using UML notations. A formal approach is considered for analyzing critical security requirements in order to analyze it. For the automated verification (model checking) process, Alloy Analyzer is considered. In the process of model checking, analysis is a form of constraint solving. Analysis can disclose subtle flaws that software architect might not have discovered until much later. Alloy is considered for modeling security patterns, because it provides a compact model that allows the verification of structural and behavioral properties of a system. Alloy makes it more logical for checking security problems in software design and provides security assurance. In this study, a model driven approach is considered, that helps to map security patterns specified in UML notation into a formal notation. Some guidelines are presented to specify the behavior of security patterns. These guidelines are useful to check the inconsistencies and ambiguities among these security patterns. The advantage of this study lies in its simple demonstration of security requirements, formal specification of these security requirements, and evaluation of security properties along with necessary security patterns. This methodology can further be extended by analyzing security patterns using ontology based modeling technique.

5 Bibliography References

References

[1] Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., and Angel, S. *A Pattern Language*. Oxford University Press, New York, 1977.

[2] Anastasakis, K., Bordbar, B., Georg, G., and Ray, I. On challenges of model transformation from UML to Alloy. *Software & Systems Modeling*, 9(1):69–86, 2010.

[3] Bayley, I. and Zhu, H. Formal specification of the variants and behavioral features of design patterns. *Journal of Systems and Software*, 83(2):209–221, 2010.

[4] Blakley, B. and Heath, C. Security design patterns. Technical Report G031, The Open Group, Apex Plaza, Forbury Road, Reading Berkshire, RG1 1AX, UK, 2004.

[5] Bordbar, B. and Anastasakis, K. UML2ALLOY: A tool for lightweight modelling of discrete event systems. In *IADIS AC*, pages 209–216, 2005.

[6] Dong, J., Alencar, P. S., Cowan, D. D., and Yang, S. Composing pattern-based components and verifying correctness. *Journal of Systems and Software*, 80(11):1755–1769, 2007.

[7] Dong, J., Peng, T., and Zhao, Y. Automated verification of security pattern compositions. *Information and Software Technology*, 52(3):274–295, 2010.

[8] Dwivedi, A. K. and Rath, S. K. Analysis of a complex architectural style C2 using modeling language Alloy. *Computer Science and Information Technology Journal*, 2(3):152–164, 2014.

[9] Dwivedi, A. K. and Rath, S. K. Selecting and formalizing an architectural style: A comparative study. In *Contemporary Computing (IC3), 2014 Seventh International Conference on*, pages 364–369. IEEE, 2014.

[10] Dwivedi, A. K. and Rath, S. K. Incorporating security features in Service-Oriented Architecture using security patterns. *ACM SIGSOFT Software Engineering Notes*, 40(1):1–6, 2015.

[11] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[12] Group, S. D. Alloy analyzer 4. `http://alloy.mit.edu/alloy4/`, 2010.

[13] Hafiz, M., Adamczyk, P., and Johnson, R. E. Growing a pattern language (for security). In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, pages 139–158. ACM, 2012.

[14] Hogg, J., Smith, D., Chong, F., Taylor, D., Wall, L., and Slater, P. *Web Service Security: Scenarios, Patterns, and Implementation Guidance for Web Services Enhancements (WSE) 3.0*. Microsoft Press, Redmond, WA, USA, 2006.

[15] Jackson, D. Alloy: A lightweight object modeling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, April 2002.

[16] Konrad, S., H. C. Cheng, B., A. Campbell, L., and Wassermann, R. Using security patterns to model and analyze security. In *In 2nd International Workshop on Requirements Engineering for High Assurance Systems (RHAS'03)*, pages 13–22. IEEE, 2003.

[17] Mikkonen, T. Formalizing design patterns. In *Proceedings of the 20th international conference on Software engineering*, pages 115–124. IEEE Computer Society, 1998.

[18] Niere, J., Schäfer, W., Wadsack, J. P., Wendehals, L., and Welsh, J. Towards pattern-based design recovery. In *Proceedings of the 24th international conference on Software engineering*, pages 338–348. ACM, 2002.

[19] Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., and Sommerlad, P. *Security Patterns: Integrating security and systems engineering*. John Wiley & Sons, West Sussex, England, 2005.

[20] Steel, C., Nagappan, R., and Lai, R. *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Prentice Hall PTR, 2005.

[21] Taibi, T. and Ngo, D. C. L. Formal specification of design pattern combination using BPSL. *Information and Software Technology*, 45(3):157–170, 2003.

[22] Uzunov, A. V., Falkner, K., and Fernandez, E. B. A comprehensive pattern-oriented approach to engineering security methodologies. *Information and Software Technology*, 57:217–247, 2015.

[23] Woodcock, J., Larsen, P. G., Bicarregui, J., and Fitzgerald, J. Formal methods: Practice and experience. *ACM Computing Surveys (CSUR)*, 41(4):19, 2009.

[24] Yoder, J. and Barcalow, J. Architectural patterns for enabling application security. In *In proceeding of the 4th Conference on Patterns Language of Programming (PLoP'97)*, 1997.

[25] Zanoni, M., Fontana, F. A., and Stella, F. On applying machine learning techniques for design pattern detection. *Journal of Systems and Software*, 103(1):102–117, 2015.

[26] Zhu, H. and Bayley, I. An algebra of design patterns. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(3):23, 2013.