MODEL DRIVEN SECURITY FRAMEWORK FOR SOFTWARE DESIGN AND VERIFICATION

by

Engin Deveci

M.S, in Computer Engineering, Boğaziçi University, 2004

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Computer Engineering
Boğaziçi University
2015

MODEL DRIVEN SECURITY FRAMEWORK FOR SOFTWARE DESIGN AND VERIFICATION

APPROVED BY:

Prof. Mehmet Ufuk Çağlayan (Thesis Supervisor)	.....
Prof. Fatih Alagöz	.....
Prof. Emin Anarım	.....
Prof. Albert Levi	.....
Prof. Semih Bilgen	.....

DATE OF APPROVAL: 11.12.2015

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Prof. Mehmet Ufuk Çağlayan for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.

I would also like to thank my family: my beloved wife Hepşen Deveci and to my daughter Derin for supporting me spiritually throughout writing this thesis and my life in general.

ABSTRACT

MODEL DRIVEN SECURITY FRAMEWORK FOR SOFTWARE DESIGN AND VERIFICATION

Information system security is receiving increasing attention every day because a security problem can cause serious financial loss or even loss of lives. Some of these security problems occur as a result of poor design practices, where important security functionality is not designed properly and is directly implemented later in the development cycle in an unmethodical way. Researchers have put a great deal of effort into defining processes and tools to design and develop more secure information systems. However, verification of the designed and developed security functionality is of utmost importance. In some cases, designs and codes also need to be formally or semi-formally verified and certified by authorities. The Common Criteria is one of the widely used universal frameworks for evaluating the security functionality of information systems. In this thesis, we propose a new framework, Model Driven Security Framework (MDSF), for the analysis, design and evaluation of security properties of the information systems. Our aim is to support information system developers and evaluation authorities who implement the higher-level Common Criteria (Levels 6 and 7) security assurance process using formal methods based on Unified Modelling Language (UML), Object Constraint Language (OCL), Promela and Spin. With MDSF, we extend UML to support security analysis and design on the UML models of the information system. In addition to UML, we use OCL in MDSF for threat identification, consistency checking among diagrams and security policy enforcement in the design model. We also propose a model transformation and model checking approach to formally verify whether the design model satisfies the security requirements listed in the analysis model.

ÖZET

YAZILIM TASARIMI VE DOĞRULAMA SÜRECİNDE MODEL GÜDÜMLÜ GÜVENLİK ÇERÇEVESİ

Uygulama ve sistemlerdeki güvenlik problemleri bilgi kaybına, finansal kayıplara ve hatta insan yaşamının kaybına varabilecek kadar ciddi sonuçlar doğurmaktadır. Bu nedenle bilgi sistemleri güvenliği her geçen gün daha fazla dikkat çekmektedir. Bu güvenlik problemlerinin büyük bir kısmı, güvenlik fonksiyonlarının tasarım aşamasında düşünülmemesinden ve geliştirme aşamasında doğrudan yazılıma eklenmesinden kaynaklanmaktadır. Halbuki, güvenlik fonksiyonlarının analizi, tasarımı ve doğrulanması çok yüksek önem derecesine sahiptir. Hatta bazı durumlarda, yazılım tasarımlarının ve kodlarının formel veya yarı-formel olarak doğrulanması ve yetkili otoritelerce sertifikalandırılması beklenmektedir. Bilgi sistemlerinin güvenlik fonksiyonlarının değerlendirilmesi için sıklıkla kullanılan standartlardan birisi Ortak Kriterlerdir. Bu tez çalışmasında, bilgi sistemlerinin güvenlik özelliklerinin analizi, tasarımı ve değerlendirmesi için yeni bir çerçeve olarak Model Güdümlü Güvenlik Çerçevesi'ni (MDSF) önermekteyiz. Amacımız, bilgi sistemleri geliştiren kişileri ve üst seviye Ortak Kriterler sertifikasyonu (EAL 6 ve EAL 7) için değerlendirme yapan yetkilileri, hazırlık ve inceleme süreçlerinde Unified Modelling Language (UML), Object Constraint Language (OCL), Promela ve Spin dil ve araçlarını kullanarak desteklemektir. MDSF ile, IT ürününün, UML modeli üzerinden, güvenlik analizi ve tasarımı yapılabilmesini desteklemek amacıyla, UML diline yeni bir profil önermekteyiz. UML'e ek olarak, güvenlik tehditlerinin belirlenmesi, farklı model seviyeleri üzerinde uyumluluk kontrolü ve IT ürününün UML tasarım modeli üzerinde güvenlik poliçelerinin doğrulanması amacıyla OCL'in kullanılabileceğini göstermekteyiz. Ayrıca, tasarım modelinin, güvenlik ihtiyaçlarını karşılayıp karşılamadığını formel olarak göstermek amacı ile model dönüştürme ve model doğrulama tekniklerini UML, OCL ve PROMELA üzerinde kullanmaktayız.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

α	Manual function to apply security requirements into the design
β	Semi-automated function to select and merge aspects into the design
U	Base UML diagram, which does not contain any security requirements
U'	UML diagram after security functions are merged

LIST OF ACRONYMS/ABBREVIATIONS

ASM	Abstract State Machine
CC	Common Criteria
CMS	Content Management System
DFD	Data Flow Diagram
EAL	Evaluation Assurance Level
EJB	Enterprise Java Beans
EMF	Eclipse Modelling Framework
FAU	Audit Security Function Family
FCO	Communication Security Function Family
FCS	Cryptographic Support Function Family
FDP	User Data Protection Function Family
FIA	Identification and Authentication Function Family
FMT	Security Management Function Family
FPR	Privacy Function Family
FPT	Protection of the TSF Function Family
FRU	Resource Utilization Function Family
FSP	Functional Specification
FTA	TOE Access Function Family
FTP	Trusted Path Function Family
GMF	Graphical Modelling Framework
IEC	International Electrotechnical Commission
IS	Information System
ISO	International Organization for Standardization
IT	Information Technology
LTL	Linear Temporal Logic
MDA	Model Driven Architecture
MDD	Model Driven Development
MDSF	Model Driven Security Framework

MDT	Model Development Tools
OCL	Object Constraint Language
OMG	Object Management Group
OSCL	Object Security Constraint Language
OWASP	Open Web Application Security Project
PGT	PROMELA Generator Tool
PIM	Platform Independent Model
PP	Protection Profile
PROMELA	Process Meta Language
PSM	Platform Specific Model
RBAC	Role Based Access Control
SECDW	Secure Data Warehouses
SFMT	Security Function Merger Tool
SFR	Security Functional Requirement
SO	Security Objective
SOA	Service Oriented Architecture
SPIN	Simple PROMELA Interpreter
SPM	Security Policy Model
SQL	Search and Query Language
SRET	Security Requirement Extraction Tool
ST	Security Target
TDS	TOE Design Specification
TIT	Threat Identification Tool
TOE	Target of Evaluation
TSF	TOE Security Function
UML	Unified Modelling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

1. INTRODUCTION

In recent years, information systems have become increasingly important in our lives. Information systems assist and complement us in almost every domain. Most of us trust these information systems with our money, assets, important information, or even our lives. Necessary precautions must be taken to prevent the damage that can occur as a consequence of a security problem in the information system. The main precaution is to make the system as secure as possible during the software development cycle.

Fixing a problem costs less and takes less time if the problem is identified early in the development cycle. Therefore, the security features of an information system need to be considered in its initial development phases. There would be a minimum number of security flaws in a software system if security requirements were handled properly during the analysis, modeled correctly during the design, implemented as error-free during development and tested thoroughly for verification. Software analysts, architects, developers and testers need sufficient tools and processes to achieve these goals. De facto standards for analysis, design, development and testing of information systems do not fully address security and hence need to be enhanced.

Stakeholders in the software development cycle require a set of common language, processes and tools that will support them in handling security requirements and features from an end-to-end perspective. Security evaluation is the most important part after the analysis and design of an information system and helps analysts or testers verify whether the designed or implemented software truly satisfies the required security functional and non-functional requirements. The Common Criteria is one of the universal methodologies that is used to evaluate information systems for security properties.

In this thesis we propose a framework called Model Driven Security Framework (MDSF) utilizing UML, OCL, PROMELA and Spin and show how we can support the

higher-level (Levels 6 and 7) assurance processes of the Common Criteria by using these languages and formal methods on the analysis and design models of an information system.

The main features of MDSF are as follows:

- extension of UML and OCL to formally support the definition of security requirements and features in the analysis and design models of the information systems,
- transformation of UML to PROMELA,
- model checking of the analysis and design models using the Spin model checker for security properties,
- alignment with the Common Criteria concepts to support the assurance process for Level 6 and 7 certifications, and
- Java (Eclipse)-based tools to support threat analysis, security requirement extraction, modeling, model transformation, and model checking.

Our approach is to create a new UML profile to support security analysis and design on the UML models of the information system. In addition to UML, we use OCL in MDSF for threat identification, consistency checking among diagrams and security policy enforcement in the design model. We also propose a model transformation and model checking approach to formally verify whether the design model satisfies the security requirements listed in the analysis model.

In Section 2, we give an introduction of object oriented software design, formal methods, and the Common Criteria. Section 3 is a literature survey describing the recent and related work about security verification, model checking UML models, Common Criteria and other related subjects. Section 4 introduces MDSF, its components and our study in detail. Proof of concept case studies showing how MDSF can be used are demonstrated in Section 5. Presented proof of concepts include the following subjects:

- Threat identification and security objective definition and security functional requirement decomposition
- Security function design weaving
- UML to PROMELA transformation and model checking dynamic security requirements
- Model checking embedded formal descriptions
- How MDSF tools and processes work end to end

The last section concludes the paper with our comments and a roadmap for future enhancements to the framework.

2. A BRIEF OVERVIEW OF THE CONSTITUENTS OF MDSF

2.1. Object Oriented Software Design

Software analysts, designers and developers have been using object oriented software design because of its maintainability, improved productivity, reusability and scalability. Object Management Group (OMG) has adopted and is managing Unified Modeling Language (UML) as a standard for visualization of object oriented software designs. The following sections give an overview of UML, its extension mechanisms, and the Object Constraint Language (OCL) for querying UML designs.

2.1.1. Unified Modeling Language (UML)

UML is a language used to for visualization of object oriented software designs including but not limited to the system's architecture, components, activities, integration, user interfaces, and how all of these will run. Software analysts and designers create diagrams to visualize the software system. UML provides all necessary language components to help analysts and designers to create these diagrams:

- Structure Diagrams: These diagrams are used to define the architecture of the software systems including its components and dependencies in between.
- Behavior Diagrams: These diagrams are used to define the functions of the software system including all the activities in and between the software components.

Structural Diagrams that can be created with UML are as follows:

- Class Diagram
- Component Diagram
- Object Diagram

- Profile Diagram
- Composite Structure Diagram
- Deployment Diagram
- Package Diagram

Behavior Diagrams that can be created with UML are as follows:

- Activity Diagram
- Use Case Diagram
- Interaction Diagram
- State Machine Diagram

In this study, we mainly used Class Diagram, State Machine Diagram and Sequence Diagram. For more information about UML, please refer to OMG UML Specifications [1].

2.1.1.1. Class Diagram. Class Diagram is a type of Structural Diagram and shows software system's classes with all the information necessary to describe it:

- Attributes
- Operations
- Relationships

An example of a class diagram is given in below Figure. In this Class Diagram we presented all the classes of the App Store software system together with class hierarchy, attributes and operations of all the classes and relations between the class.

2.1.1.2. State Machine Diagram. State Machine Diagrams are used to represent the functional behavior of a class in the software system. States can be basic or nested where a detailed state machine is abstracted in a single nested state. Transition from one state to another is triggered by internal or external events. State change happens

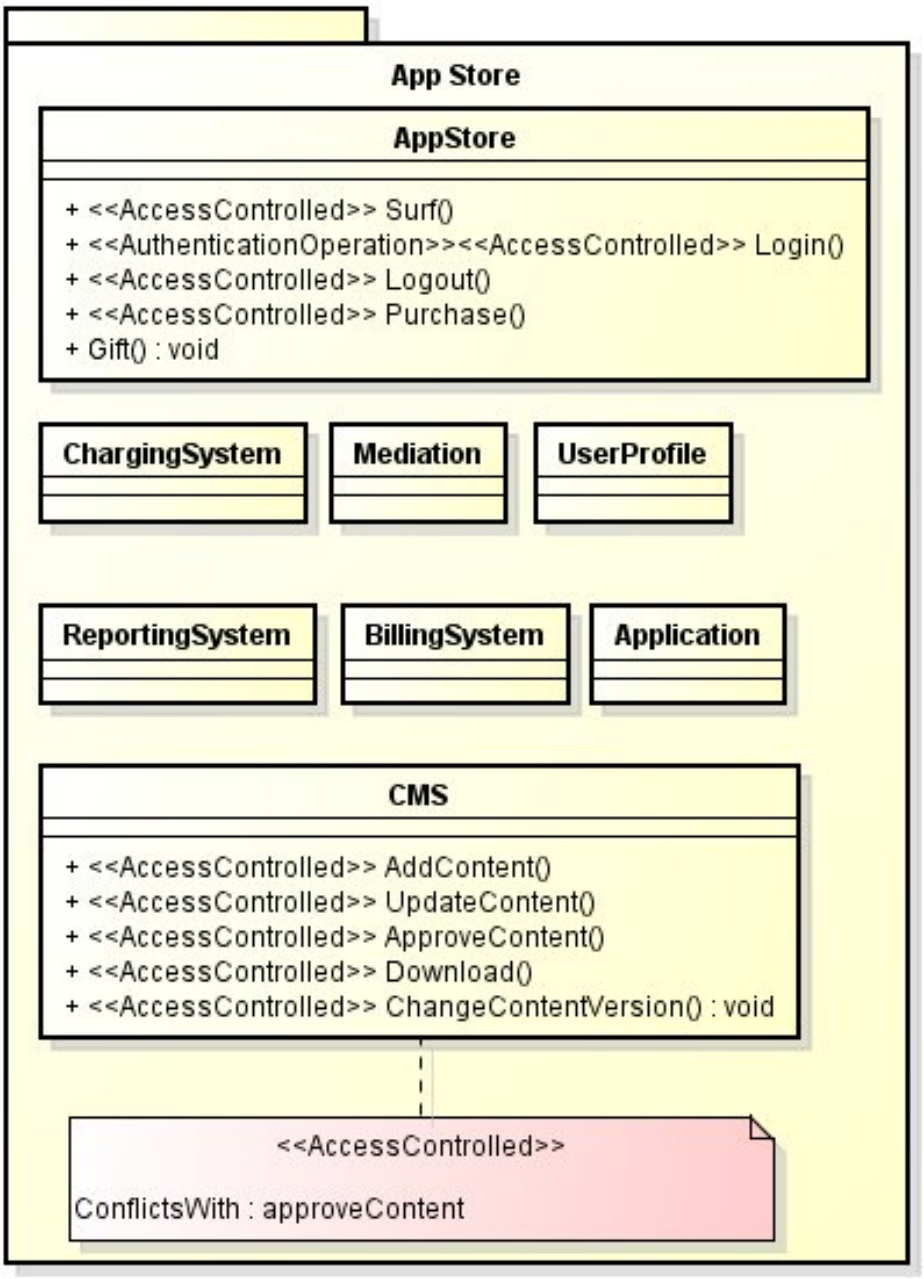


Figure 2.1: Class Diagram Example

if the triggering event exists and the guard condition of the transition is satisfied.

An example State Machine Diagram is given in below Figure. In this State Machine Diagram, we presented all the states of the "Prepaid User" class together with all the internal and external triggers and associated state transitions.

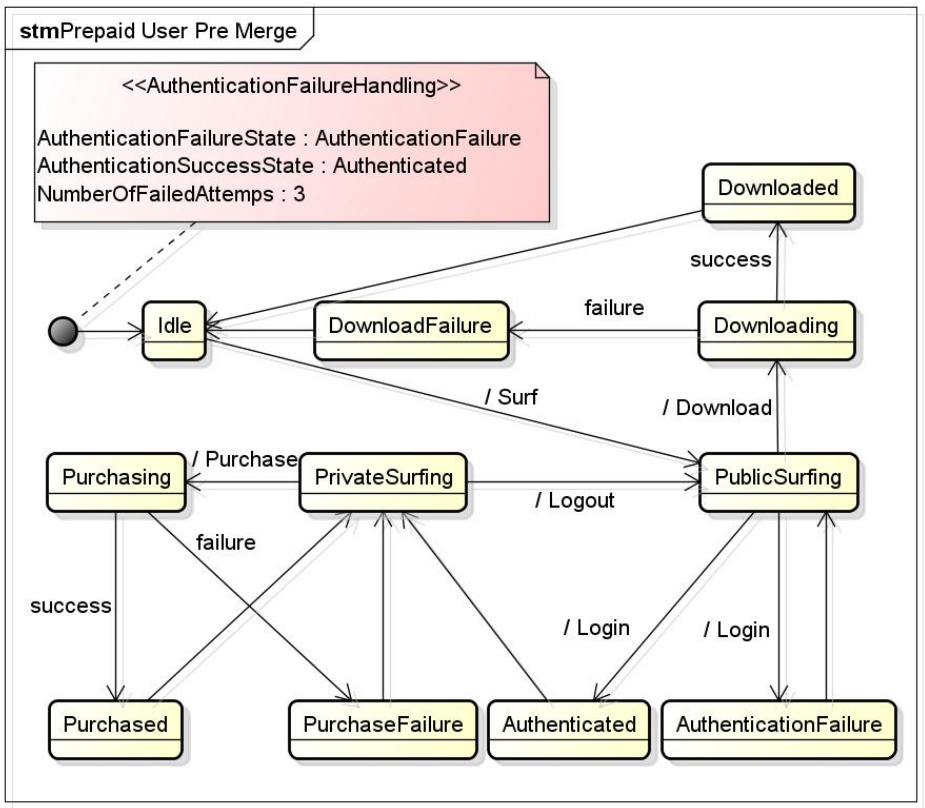


Figure 2.2: State Machine Diagram Example

2.1.1.3. Sequence Diagram. Sequence Diagrams show the interaction between the objects in the software system. These interactions between the processes of the objects are presented in a time sequence. Usually, the sequence diagrams are used to explain the behavior of all of the objects involved in a specific use case of the software system. The interactions are triggered by message passing between the processed by operation calls or function returns. These interactions can be synchronous or asynchronous.

An example sequence diagram is given in above Figure. In this sequence diagram we presented the sequence of interactions between the objects together with the conveyed information within each interaction.

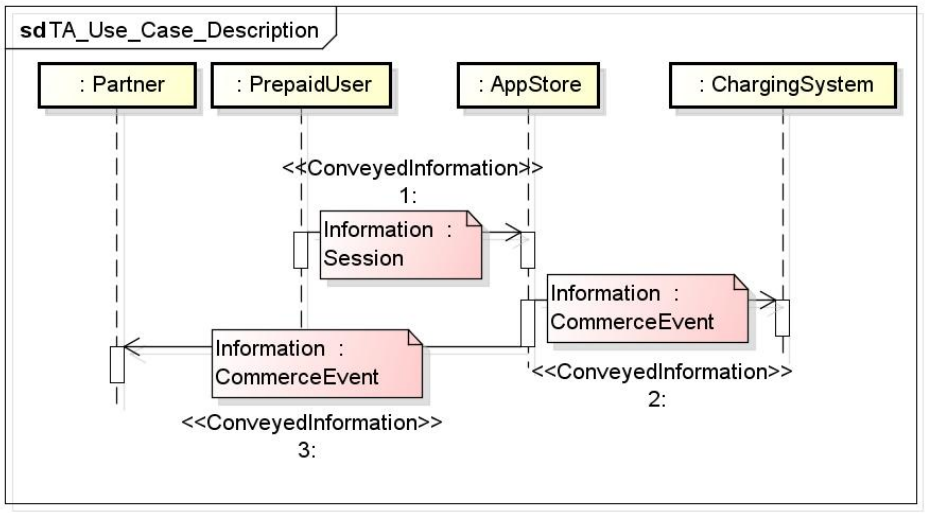


Figure 2.3: Sequence Diagram Example

2.1.2. Extension Mechanisms – UML Profiles

Whenever it's needed to extend UML for particular domains or platforms, two different extension mechanisms can be used:

- Using Profiles to customize UML
- Adding new meta-class and meta-relationships to the meta-model and hence creating a new UML dialect.

In this study, we have selected to use Profile mechanism since it is supported by most of the UML design tools in the market today. UML profiles are recommended in the following circumstances:

- To create a new terminology or domain model such as security
- To create a syntax for constructs that do not have a notation
- To complete the semantics which is currently left unspecified in the UML meta-model
- To add new semantics into the UML meta model
- To be able to specify constraints that restricts the way meta-model and constructs can be used.

- To add additional information that will guide and help the model transformation

In this thesis, by applying a profile to UML, we are refining the semantics of UML elements for security domain and create a level of formalism, where we avoid contradicting semantics for certain elements and diagrams.

A profile is a collection of stereotypes, tagged values, all together refine the semantics of UML for a specific domain. Stereotypes are defined using "<<>>" around a keyword. When applied to a UML model element such as a class, an attribute or an operation, the stereotype gives a new meaning to the element and hence creates a specialized model element that can be used as a building block while creating UML diagrams for a specific domain or platform. In example, designer can create a Database class, by applying a "<<Database>>" stereotype to UML class model element. Each stereotype can have 0 or more attributes, called tagged values. These tagged values allow the designer to add extra meaning to a stereotyped element with the help of key value pairs. An example UML Profile is given in below Figure.

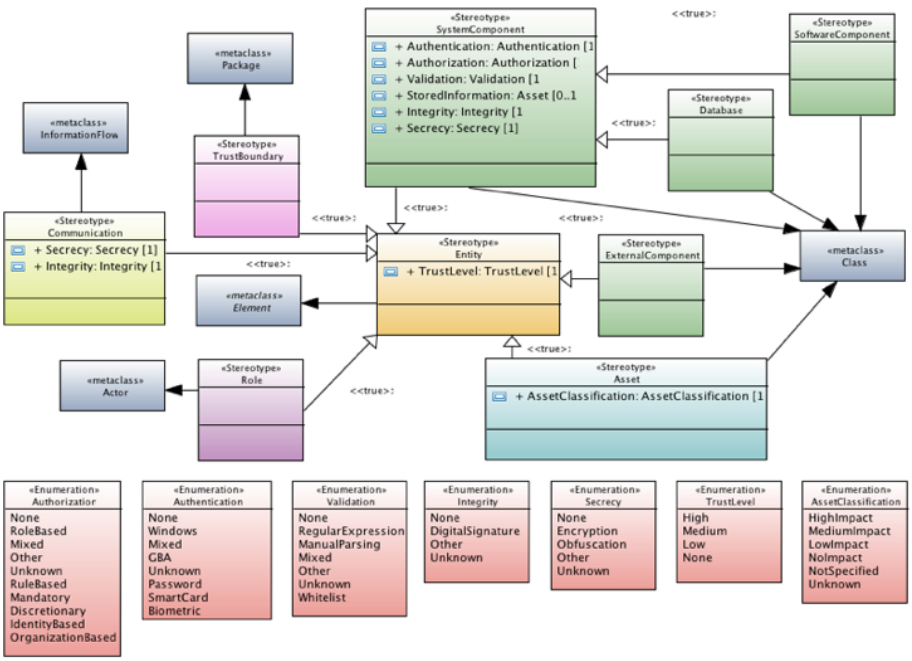


Figure 2.4: UML Profile Example

For more information about UML Profiles, please refer to OMG UML Profiles and Related Specifications [2].

2.1.3. Object Constraint Language (OCL)

There's always a need to specify additional constraints on the UML model to resolve ambiguities and refine the model. Before OCL, these constraints were written on the natural languages. In order to write unambiguous constraints, people started to use formal languages. However, formal languages required a mathematical background and it was difficult for an average analyst or designer to use. OCL has been developed as a solution to this problem. OCL is:

- a formal language
- easy to read an write, does not require a mathematical background
- a specification language, does not change any values or state of the model
- not a programming language
- typed and well formed language
- evaluated instantaneously and hence model cannot change during evaluation

OCL is written as statements. Each statement contain below elements:

- Context: Defines the UML model section where the statement will be valid, or applied.
- Properties: Defines an element under the context
- Operations: An arithmetic or set based operation to manipulate or qualify a property
- Keywords: Conditional or logic operators

In this thesis, we use OCL as a formal specification language for UML to define constraints and object query expressions on model diagrams. These constraints are used for model correctness and conformance as well as threat identification and security policy enforcement.

An example OCL Statement is given in below Table.

Table 2.1: Example OCL Invariant for Checking UML Stereotypes

```
self.allOwnedElements() ->
    select ( a | a.oclIsTypeOf( uml::Class ) ) ->
    forAll ( b | b.getAppliedStereotypes().name ->
        intersection( Set (
            'SoftwareComponent'
            'Database',
            'Asset' ) ) -> size() > 0 )
```

For more information about OCL, please refer to OMG OCL Specifications [3].

2.1.4. Model Driven Architecture

Model Driven Architecture (MDA) is an initiative driven my OMG. It aims to separate the application logic from the underlying platform technologies.

In MDA, application and business logic is designed using a modeling standard such as UML using domain specific extensions. It's completely independent of the underlying platform technologies such as web services, .NET, J2EE, CORBA, and others. This platform independent model (PIM) is then transformed into a platform specific model (PSM) using MDA tools and transformation mappings.

In this study, we employed MDA approach to convert software system design specifications prepared in UML with domain specific profiles to PROMELA in order to use in model checking.

For more information about MDA, please refer to OMG MDA Specifications [4].

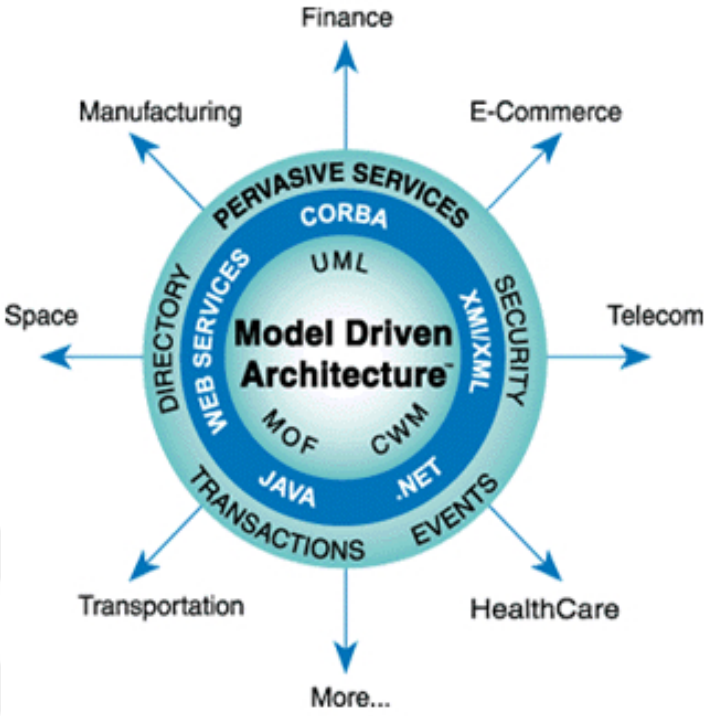


Figure 2.5: OMG Model Driven Architecture

2.1.4.1. Model Driven Design and Development with Open Source Tools. In this study, we use Eclipse Modeling Project and Model Development Tools since they are open source, widely used and have community support. The main tools used in the thesis are:

- OCL: Implementation of OCL for EMF-based models
- UML2: EMF based implementation of UML2.x metamodel
- UML2 Tools: GMF based editors for viewing and editing UML models
- Papyrus: UML2 editors supporting the standard and the extension mechanisms such as profiles

For more information about Eclipse platform, please refer to Eclipse MDT web site [5].

2.2. Formal Methods for Verification of Software Designs

Formal Methods are a special kind of mathematical techniques for formal specification, design, development and verification of software and hardware systems. In this thesis, we use formal methods mainly for verification of software systems for security requirements.

There are two categories of formal methods used in automated verification of correctness:

- Automated Theorem Proving: Given the system description, logical axioms and inference rules, tries to generate formal proof of correctness.
- Model Checking: Executes an exhaustive search on all possible states the software system can be in and verifies the given properties.

We use model checking approach to verify the security requirements on the given software system design model. The next section gives an overview of model checking approach.

2.2.1. Model Checking

Model Checking is used for formally verifying correctness properties in finite state software or hardware systems. Given a system model, model checking exhaustively searches if the model satisfies the given requirements or properties.

For the exhaustive search, model checking requires the model and the properties to be written in a formal/precise mathematical language. In this study, for model checking we have used Spin Model Checker tool since it's freely available and there's strong community support. Spin Model Checker uses Promela as the precise specification language. For more information on Spin Model Checker and Promela language please refer to Principles of the Spin Model Checker book [6].

2.3. Standards for Security Certification (The Common Criteria)

Security aspect of the IT products is one of the most important buying criteria for the enterprises. It's easier for the enterprise to decide if the IT product has a security certification for the particular security properties.

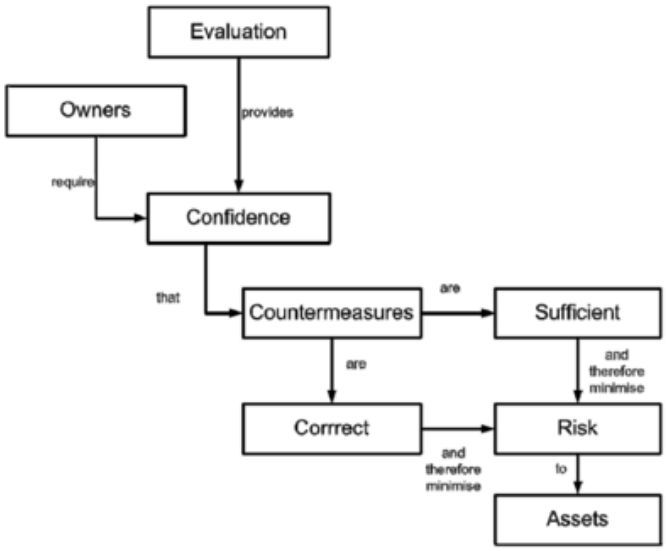


Figure 2.6: Purpose of Certification/Evaluation

Certification creates confidence in the IT product that all the correct and sufficient countermeasures are considered and implement to avoid the risks to the assets. The Common Criteria (The CC, ISO/IEC 15408) [7] [8] [9] is a well known international standard for IT product security evaluation and certification. The CC has certificate authorizing members in seventeen countries including Turkey. Products are evaluated in the independent licensed laboratories to determine the fulfillment of particular security properties. There are seven evaluation assurance levels (EALs) in the CC and EAL 7 is the most difficult one to achieve due to its requirements on formal modelling and formal verification of the security features. This is why, there are only a few certified products on EAL 7 level. In this study, we're proposing a new methodology to ease the preparation and evaluation processes for the CC certification. In this new methodology, we use UML and OCL to model the functional specification and security policies of the IT product. And later on, we formally verify the security policies on the functional specification using the SPIN model checker.

There are three sections of CC:

- Part 1 (ISO/IEC 15408-1): It describes the general CC structure and provides background information [7].
- Part 2 (ISO/IEC 15408-2): Contains common security requirements for IT products and is used as guidance during the formulation of security requirements [8].
- Part 3 (ISO/IEC 15408-3): Contains assurance requirements and used as guidance during definition of required assurance levels [9].

In this study we will use only Part 2 of The Common Criteria which contains general functional security requirements related to security audit, communications, cryptographic support, user data protection, identification and authentication, security management, privacy, protection of the TSF, resource utilization, TOE access and trusted path/channels

Evaluation according to the Common Criteria has been required by the U.S. government for acquisition of information systems dealing with information security since 2002 [10]. Countries have legal certification authorities responsible for certification and these organizations assign evaluation task to independent laboratories. As an example, Turkish Standards Institution (TSE) in the name of Turkey is the national Common Criteria certification body since 2003 and certified TUBITAK UEKAE Common Criteria Test Center (OKTEM) as the independent test laboratory.

2.3.1. The Common Criteria Concepts

In the Common Criteria assurance model, the IT product is verified in two angles:

- Security Requirements: In this part, completeness and correctness of identified security threats, security objectives and security requirements are checked and verified. If any of these are missing, incomplete or incorrect, IT product's security properties will never be complete or correct.
- Design and Development: In this part, testers check if the IT product's designers

and developers have considered all the security objectives and security functional requirements during the design and development process. Below cases can be identified with this evaluation:

- Wrong design decisions
- Wrong or partial interpretation of security functional requirements
- Wrong or partial development of security functions

Verification steps of the IT product security features are as follows:

- Analysis and verification of process and procedures
- Verification of process execution
- Verification of conformance of different level of design models of the IT product
- Verification of the security functional requirements on the design model of the IT product
- Verification of proofs
- Analysis of the documentation
- Functional tests of the IT product and analysis of test results
- Independent functional tests
- Analysis of security flaws
- Penetration tests

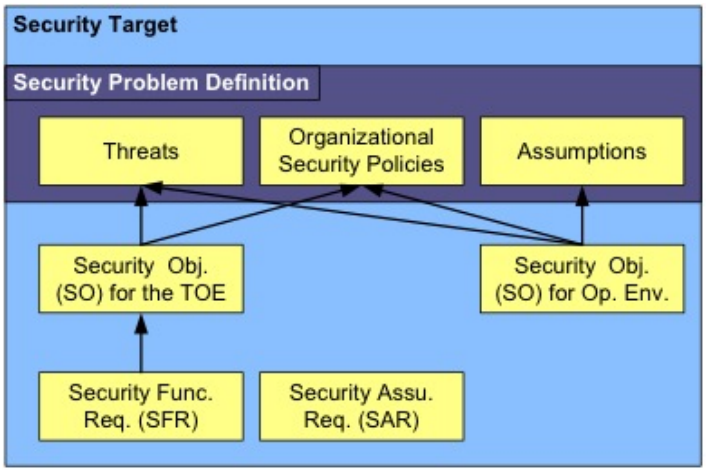


Figure 2.7: The Common Criteria Concepts

Relations between the threats, security objectives and security functional requirements are represented in Figure 2.7. Security objectives are high-level requirements, which aim to mitigate threats described in the security problem definition. Those high-level security objectives are decomposed into security functional requirements. All of this information needs to be documented in a Security Target (ST) document. This document describes what will be evaluated as the target of evaluation (TOE). Full contents of the ST document is explained in Figure 2.8. The related sections of the ST document for this study are:

- Description of the security problem
- Security objectives
- Security requirements

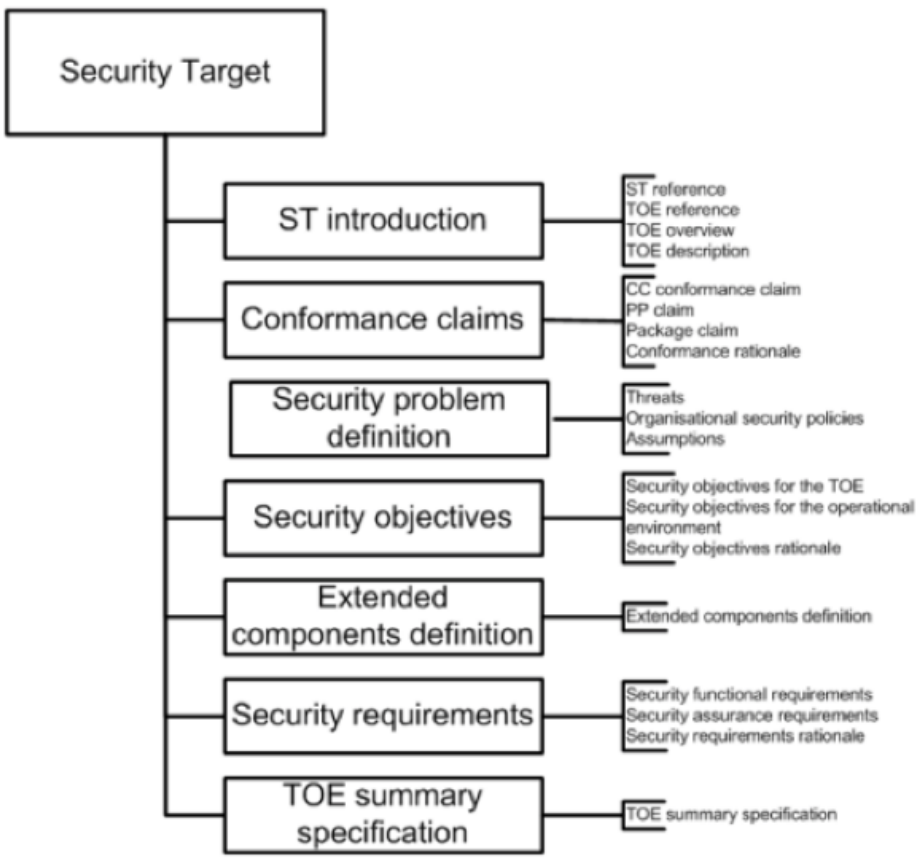


Figure 2.8: Contents of the Security Target

2.3.2. Certification Process and the Role of Formal Methods

There are seven Evaluation Assurance Levels (EAL) defined in the Common Criteria evaluation process. Higher assurance levels require more effort to design and to evaluate. Our framework will assist designers and evaluators for higher level of assurances:

- EAL5: Semi formally designed and tested
- EAL6: Semi formally verified design and tested
- EAL7: Formally verified design and tested

In the Common Criteria, especially in EAL6 and EAL7, there are different types of documents that needs to be prepared [11]. Figure 2.9 explains these assurance topics and assurance requirements.

IT product provider shall provide assurance for IT product's security problem, security objectives, security properties, software/system design, software code and conformance of all with each other.

The level and complexity of assurance requirements also increase as the assurance level increases in the Common Criteria certification. In the EAL7 assurance level, below main requirements are expected:

- ADV_FSP.6: Functional specification of the software shall be formally represented. Also the relation between the formal functional specification and security functional requirements shall be explained.
- ADV_SPM.1: A formal security policy shall be provided. And, for each security policy, satisfied security functional requirements shall be listed. The conformance between the security policy and functional specification shall be formally verified.
- ADV_TDS.6: A formal model shall be provided as the design model of the software. And conformance between the functional specification and the design model shall be verified formally.

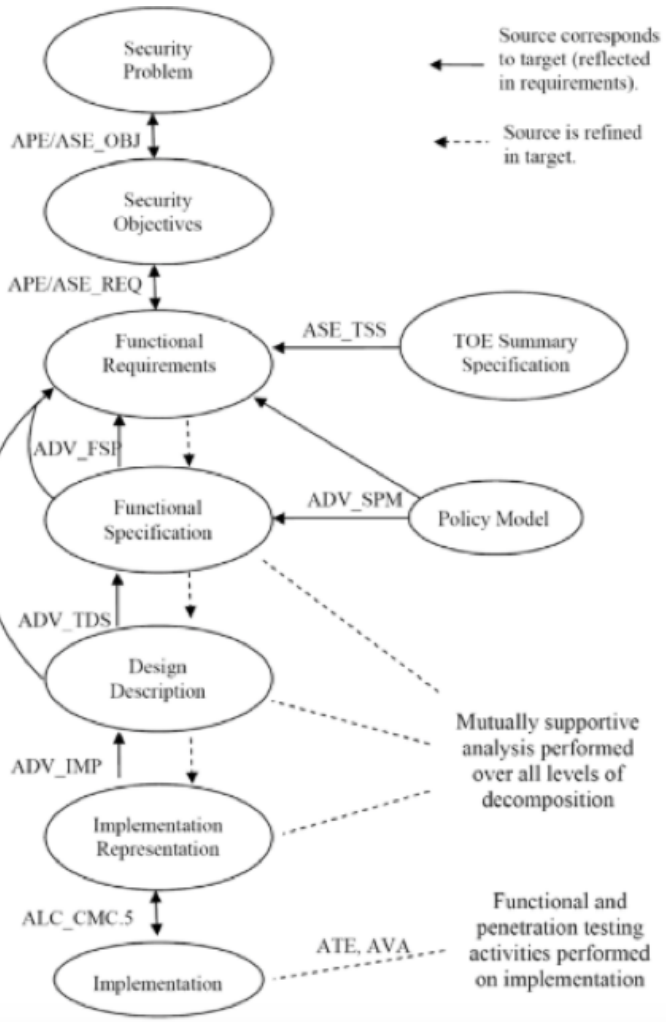


Figure 2.9: The CC Certification Process

In MDSF, in order to ease the certification preparation, analysis and verification, we propose below methods:

- Identification of security threats using the UML analysis model
- Extraction of security objectives from the identified security threats
- Decomposition of security objectives to security functional requirements
- Design of functional specification with UML, satisfying the security functional requirements
- Preparation of the security policy with OCL
- Formal verification of the security policy on the functional specification and design model

3. STATE OF THE ART IN SECURE SOFTWARE DESIGN AND VERIFICATION

3.1. Secure Software Development Lifecycle

Although there is significant need for an end to end software development methodology taking security into consideration in all of the phases, lots of existing studies are usually concerned with security in a smaller scale. There are only a few studies to increase the reliability and robustness of software using a more controlled software life cycle including but not limited to risk management frameworks, code reviews, security testing and abuse case development [12]. Microsoft also contributed to this area by introducing their secure development life cycle which helps to minimize security defects in software [13]. They also developed a tool based on data flow diagrams to detect the security threats early in the software development cycle. Noopur summarized most of the secure software development life cycle approaches in a technical report [14]. This approach aims to continuously remove the security vulnerabilities at analysis, design and implementation. Villarroel et al. and Cabot et al. both prepared surveys for different methodologies proposed to integrate security into the phases of software development [15] [16]. In their surveys, they claimed that none of the available tools and methodologies can be used to completely specify security aspects of the software. Still, all of these approaches contain valuable methods that can be used for a more complete methodology for secure software development.

3.2. Using UML and OCL for Security Analysis and Design

Handling security requirements using UML during analysis and design of information systems has been researched for a long time. There are several proposals for embedding security into the software design. Some researchers have used UML without any modifications [17], though others propose extensions to UML that result in more expressive power and formalization during security design. One of the prominent

proposals is UMLsec [18]. UMLsec is a UML profile created with stereotypes that help software analysts and architects include security requirements in the UML analysis and design models. Requirements such as fair exchange, secrecy, confidentiality, secure information flow and secure communication links can be specified using these stereotypes. Jurjens also proposed a formal method for the analysis of the security features of the models. In this approach, it is possible to model check some of the UML diagrams using the proposed framework [19] [20].

Peralta et al. [21] also extended UML with some security-related stereotypes. They proposed to use this new UML profile for automated test case generation from the UML model. They defined the security stereotypes based on the Open Web Application Security Project (OWASP), which lists the top ten most critical web application vulnerabilities, including buffer overflow, connection flooding, encryption, access control, and Search and Query Language (SQL) injection. Goudalo and Seret [22] worked on the security of Information Systems (ISs) and proposed another UML profile with secrecy, trust and role based access control (RBAC) stereotypes. Villarroel et al. [23] worked on data confidentiality problems in the conceptual modeling of data warehouses. They defined a new UML profile named Secure Data Warehouses (SECDW) using new stereotypes and tagged values. They also extended OCL with new data types. This new profile is used to specify security levels on data, compartments and user roles on the model. Sohr et al. [24] also worked on RBAC. They defined an authorization engine and a validation framework for RBAC policies in web services.

Pavlich Mariscal et al. [25] stated that UML does not have formal and explicit support for access control and proposed the use of new diagrams for access control aspects such as role-based, discretionary and mandatory access controls. Lodderstedt et al. [26] proposed another UML profile, SecureUML, with new stereotypes and tagged values to annotate UML models with information relevant to access control. They also showed that it is possible to generate Enterprise JavaBeans (EJB) applications with fully configured RBAC, including role definitions, method permissions, user role assignments and authorization constraints using some model driven architecture (MDA) tools and methodologies. Poniszewska Maranda [27] proposed an approach in which

RBAC-based security constraints can be specified in UML designs using OCL without any modifications. Fernandez Medina et al. [28] proposed an extension to the OCL language to solve the problem of security in database design. Proposed Object Security Constraint Language (OSCL) is used to specify security constraints in class diagrams. There are also several studies on the authentication properties of software models.

Mehr and Schreirer [29], discussed the security requirements in service oriented architectures and stated that none of the available approaches are sufficient for the treatment of message security during modeling. They discussed UML Profiles for security, UML templates, and UML and OCL without modification. They proposed a new approach where security concerns are separated from the model with a message security policy file and integrated this policy with the model using a slightly extended version of OCL. Borek et al. [30] also proposed an OCL based approach to formalize the security policies on a UML based model. They also converted the OCL statements into algebraic notation and verified the security requirements using an electronic ticket system as a case study. Lincke et al. [31] proposed using misuse case diagrams for security analysis on UML diagrams. They focused on system-level architecture instead of focusing on just the software architecture and mentioned their proposal can be used not only for software design, but also for audit, testing, security planning and security education. Muntean et al. [32] proposed a methodology to introduce information flow security constraints into UML statecharts and also into the C source code of the software in order to enable automated analysis of the design and the code in information flow checkers. Katkalov et al. [33] also worked on specification of information flow constraints on UML models and proposed a model driven approach to automatically generate a formal specification and code.

3.3. Model Driven Approaches for Secure Software Design and Verification

Buchholtz et al. [34] proposed an approach, "For-LySa", which uses a new UML profile to easily convert UML diagrams to the process calculus LySa. In this study, an external analysis tool is used for the authentication analysis on the process calculus. Peterson et al. [35] defined a new UML profile that can be used to embed security re-

quirements into the UML class diagrams. They included a stereotype named security package that includes a risk factor, security title and a descriptor that will guide developers during the development of security aspects. Breu et al. and Alam et al. [36] [37] proposed SECTET and SECTET-PL to model security requirements with respect to service-oriented architecture (SOA).

In addition to UMLsec [18] [19] [20], other approaches aim to verify security designs using model checking. In the SOCLe project [38] [39], the authors proposed a methodology for converting UML state diagrams to Abstract State Machine (ASM) and model-check military access control policies. Börger et.al [40] also worked on formalizing UML state machines using ASMs. Mezei et. al. [41] shows a way to formalize OCL constraints using ASM technique. Moebius et al. [42] also proposed a UML profile to include security properties into the model. They provided the translation of these models to Abstract State Machines (ASM), which can be used for the verification of security properties.

Zisman [43] worked on the security of peer-to-peer applications and proposed a framework for modeling these applications using UMLsec. That framework includes some translators that can convert design models into PROMELA and then uses Spin Model Checker [44] for model checking some properties. Siveroni et al. [45] extended this approach for more generic security properties such as availability, integrity and confidentiality using a property specification language (USVF). Munante et al. [46] proposed an extension to UMLsec to model access control security policies and assess it to guarantee if they are well-formed, to analyse potential conflicts, and to simulate it. Kaliappan et al. [47] stated that model driven design and development (MDD) can be used to develop communication protocols. They also examined how UML state diagrams can be translated into PROMELA and can be checked with Spin Model Checker. Mana et al. [48] have defined their ultimate future goal as the automatic generation of executable systems with fully configured security infrastructures from the business process model. Their current work is focused on the definition of a UML profile for security requirements, the extension of UML tools in order to support these profiles. Basin et al. [49] used SecureUML for specifying access control requirements

of software systems and proposed to use MDA for automated creation of security infrastructures from models in the future. Mouheb et al. [50], from an aspect oriented view, separated the base software model and security aspect model. In their study security aspect model is created from a library which is managed by security experts. They also proposed a methodology to combine these two models using a model weaving mechanism.

Borek et al. [51] proposed a model-driven approach called SecureMDD to abstract software specifications for optimized model checking of the security constraints. They used Avantssar for model checking and demonstrated their approach for a smart card example. Ledru et al. [52] converted a SecureUML model into B language to validate the functional model and to verify the permission rules.

3.4. Analysis and Design Approaches for the Common Criteria Certification Process

Conformance to security standards is important for security verification. One of the most detailed studies in this area was conducted by Mellado et al. [53]. They proposed a process for engineering security requirements. In their paper, they presented a theoretical background but did not propose any tools to support their approach. Ware et al. [10] proposed a different methodology to consider security earlier in the design phase of the software development cycle. In their study, they used the Common Criteria to elicit security requirements from the cases that were used. They used actor profiles in the cases, derived threats based on them, mapped these threats to security objectives and later decomposed the threats to the security requirements. Saeki et al. [54] proposed a similar methodology to elicit security requirements, also using a Common Criteria-based approach to define a semi-automatic elicitation methodology. Morimoto and Cheng [55] worked on protection profiles that are certified to satisfy the Common Criteria properties. They proposed model protection profiles using UML so that designers can easily specify and reuse security aspects. They also stated that these protection profiles can be verified using theorem proving with Z.

Beckers et al. [56] proposed a methodology based on UML4PF (UML for Problem Frame) and OCL to support security reasoning, model validation and document generation for the CC security requirements. Aoki et al. [57] also used UML for for security requirement specification and transformed them to finite automaton to be verified in UPPAAL model checking tool.

3.5. Inadequacies of Current Approaches to Support the Common Criteria Certification

According to the Common Criteria certification guidelines, there are seven main areas that can be focused to ease the preparation and certification process:

- Security problem definition
- Security objectives
- Security requirements
- Formal security policy specification
- Functional specification and design model
- Conformance between different models
- Formal verification of security policy and design model

None of the current approaches provide an end to end methodology and tools to support software analysis, design and verification aligned with the Common Criteria assurance process. There are some studies to semi-automatically identify security threats and decompose them to security objectives and security requirements. However, these approaches depend on some tools and languages on which the software industry does not have any competence.

The major problem with the Common Criteria certification process is its complexity, lack of competence and long preparation timelines and high costs. In order to ease the whole process, tools and languages shall be selected such that software industry will feel comfortable using them. This is why, instead of proprietary mechanisms, de facto industry standards, UML and OCL shall be used to the possible extend.

Some proposals use proprietary languages, some others propose extending the meta-model of UML and OCL. These approaches will only increase the complexity and since there'll be no tool support, will increase the time spent on the preparation process. Several studies propose new UML profiles instead of meta-model extensions to model security properties of a software. Some of these proposals are incomplete since they only focus on a single security domain such as access control. The rest is more complete, but they don't have any alignment with the Common Criteria security function families and components.

OCL is a very strong de facto language for defining queries and invariants. There are several available commercial and community tools for the execution of OCL on the UML models. It's a very good fit for conformance checking between various design models, definition of security policy for the static security requirements and verification of static security requirements on the design model. However, there are only a limited number of studies recommending OCL for such activities and they propose using OCL with some modifications and only on a smaller context such as access control or secure databases.

4. MODEL DRIVEN SECURITY FRAMEWORK FOR SOFTWARE DESIGN AND VERIFICATION

MDSF is proposed to assist software analysts, architects and security engineers during the system development cycle to create a more secure and robust system. The main focus of the framework is on the analysis and design phases (Figure 4.1) to formally verify whether the design model satisfies the listed security requirements in the analysis model.

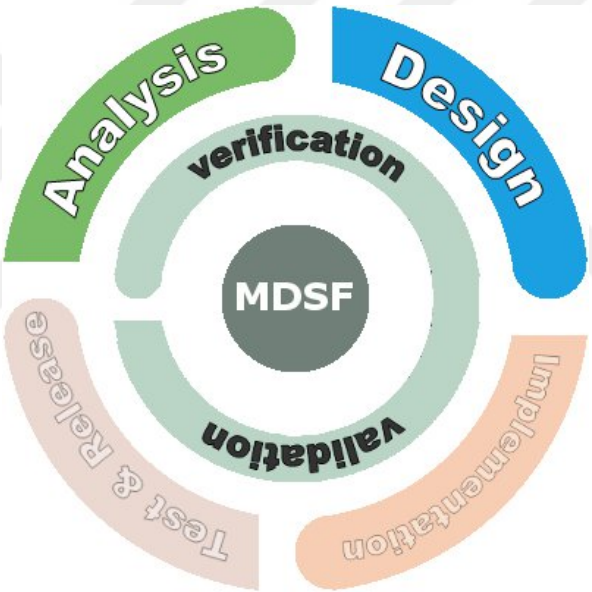


Figure 4.1: Main MDSF Cycle

Having a complete specification of the security requirements is the starting point for creating a secure system. Therefore, MDSF helps the analysts to create an analysis model that includes security properties to identify additional threats from the model and transform these threats into additional security requirements in the analysis phase.

In the design phase, MDSF helps the designer to create a design model that includes the security features of the system, merge pre-verified security functionality into the design, and formally verify whether the final system design satisfies the security requirements identified in the analysis phase. Analysis and design models prepared with the MDSF can be transformed automatically into source code and test cases for the

implementation and testing phases. Model-driven code generation (implementation stage) and test case generation (testing stage) are very wide topics that need special attention and are not covered in our study.

4.1. Structure of MDSF

MDSF is structured as three layers. The MDSF layers, from bottom to top, are the Data Layer, the Application Layer and the Business Layer (Figure 4.2).

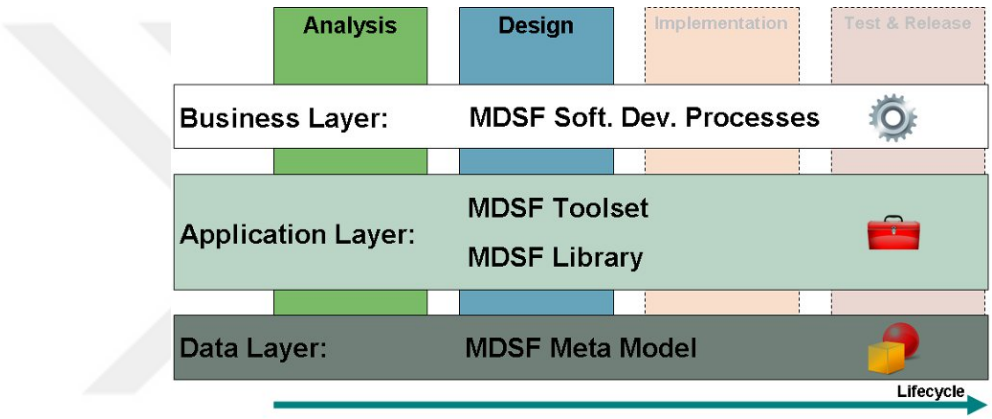


Figure 4.2: MDSF Structure

At the bottom, the Data Layer contains the MDSF Meta-model, which is the foundation of MDSF. The MDSF Meta-model defines the semantics of MDSF and explains its relation to the Common Criteria security concepts. The Application Layer contains the MDSF Toolset and Library, which collaboratively support the software development processes. The MDSF Toolset provides tools that support security requirement analysis, functional design, and verification phases. The MDSF Library contains the Common Criteria repository and security function repository that are used by the MDSF Toolset for the Common Criteria alignment and security function weaving during the design phase. The top layer is the Business Layer, which governs the steps of the MDSF. The Business Layer provides a structured approach to achieve a higher quality security design by describing the tasks to be executed at each step of the software development phases. These tasks are the guidelines on why, how and when to use the MDSF Meta-model, Toolset and the Library. In the following sections, detailed descriptions of the MDSF Meta-model, Library and Toolset are given.

4.2. MDSF Meta-model

The MDSF Meta-model defines the entities and their interactions within the MDSF. The MDSF Meta-model is also used to create a common understanding of the modeling concepts. Unlike other security frameworks, the MDSF Meta-model is aligned with the Common Criteria [8] to make it useful during the security assurance process (Figure 4.3).

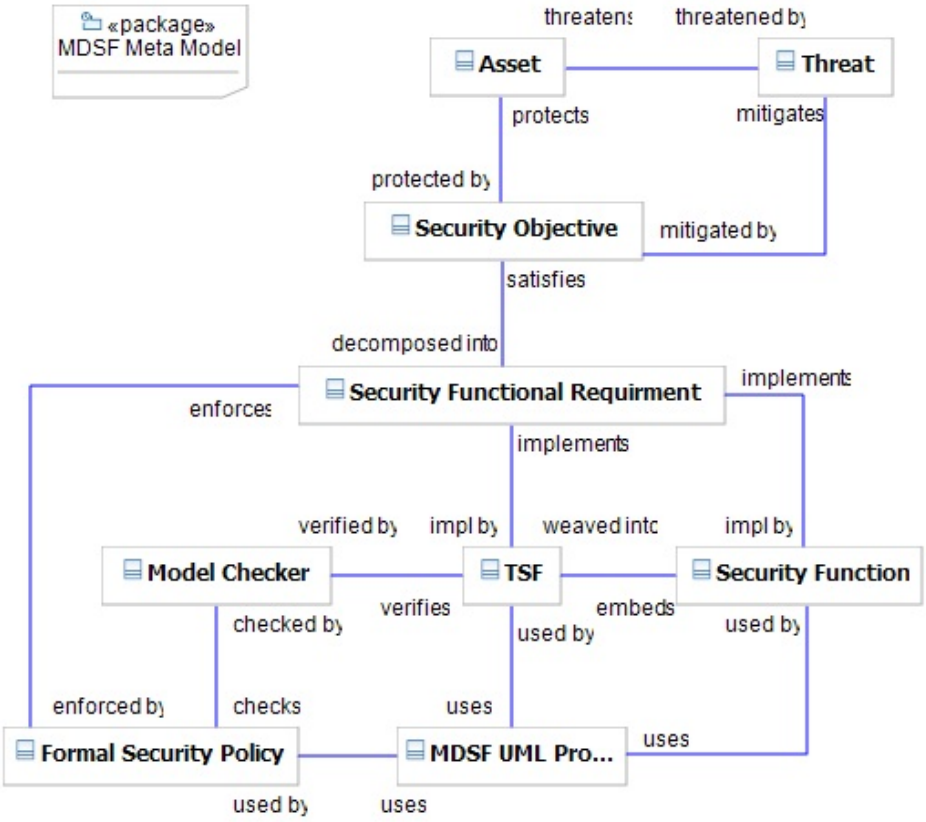


Figure 4.3: MDSF Meta-model as a UML Class Diagram

The way in which MDSF Meta-model is aligned with the Common Criteria is described below (Table 4.1). The Common Criteria provides a common set of requirements for the security functionality of software systems and supports an evaluation process to establish a level of confidence that the security functionality of the software system satisfies the security requirements. Common Criteria addresses the protection of "assets" from "threats" regarding the assets' confidentiality, integrity and availability. Common Criteria uses the concept of a Security Target (ST) that contains assets, threats to those assets and countermeasures in the form of "security objectives" (SOs).

ST needs to detail SOs in terms of "security functional requirements" (SFRs). Software systems to fulfill these SFRs are defined by the "Target of Evaluation" (TOE) Security Function (TSF) design model. Some pre-verified "security functionality" can also be woven into the TSF design model. The overall purpose of MDSF is to formally verify whether the TSF design model satisfies the security policies and thus contributes to the final Common Criteria assurance of a software system in terms of security.

Table 4.1: Description of Meta-model Components

Component Name	**Component Description**
Asset	Information stored, transferred and processed within the boundaries of the system is classified as asset.
Threat	Threat is the possibility that vulnerability may be exploited to cause harm to a system, environment, or asset.
Security Objective (SO)	Security Objective is a statement of intent to counter identified threats and/or satisfies identified organization security policies and/or assumptions (The Common Criteria).
Security Functional Requirement(SFR)	Security Function Requirement is a translation of the security objectives for the TOE into a standardized language (The Common Criteria).
Target Of Evaluation (TOE) Security Functionality (TSF)	TSF is a set consisting of all hardware, software, and firmware of the TOE that must be relied upon for the correct enforcement of the SFRs (The Common Criteria). In the scope of MDSF TSF is also used for the "TSF Design Model".

Continued on next page

Table 4.1 – continued from previous page

Component Name	Component Description
Security Aspects	Security aspect is a design model block, which is pre-defined to satisfy a set of common security functional requirements, and is able to easily integrate into a system model.
MDSF Profile	UML Profile to support MDSF processes.
Formal Security Policy	A set of rules describing specific security behavior enforced by the TSF and expressible as a set of SFRs.
Formal Verification	Method to prove, disprove that TSF really enforces specified formal security policies.

MDSF suggests that security-driven software analysis and design must be strongly aligned with industry view and practices. Although UML is currently the de facto industry standard, UML is not sufficient to model security requirements and security functions. As previously discussed by Villarroel et al. [15], Jurjens [18], and Peralta et al. [21], one of the most appropriate ways to model security requirements and functions is to use the standard extension mechanisms of UML, such as profiles. It is possible to define any type of security requirement by extending UML with a profile that contains security stereotypes and tagged values. Analysts and designers use stereotypes to extend the UML vocabulary to contain new model elements for specific domains such as security. Tagged values are properties of the stereotypes and add further detail to the stereotyped model element. The application of stereotype and tagged values to a class in a UML class diagram is described in Figure 4.4.

The MDSF UML Analysis Profile and MDSF UML Design Profile contain a stereotype set and the underlying meta-model that describes the relationships between

Figure 4.4: Stereotype and Tagged Value

these stereotypes. Creating a TSF design model with security features to satisfy selected SFRs is possible with MDSF UML Profile.

4.2.1. MDSF UML Analysis Profile

A subset of stereotypes in MDSF UML Analysis Profile is given in Figure 4.5. Threat Analysis Package and its contents are inspired and extended from the work done by Wang et.al., and [58] Abi-Antoun et.al. [59]. In these papers authors used specialized Data Flow Diagrams (DFDs) and extended them with some properties. Their aim is to use DFDs with some special properties to identify possible threats in the system. In the Threat Analysis Package, work done by these authors is extended with the addition of new stereotypes, and a specialized UML profile is created. Instead of DFDs, UML diagrams are used and threat identification rules are created in pure OCL.

Analysts and designers use the UML Analysis Profile to specify system components, critical assets, communication protocols, roles, entities and their security properties in the UML analysis model. An annotated version of the analysis model is created by applying the stereotypes and tagged values from the MDSF UML Analysis Profile to the original analysis model. The MDSF Toolset uses the annotated analysis model to semi-automatically identify possible threats for the model and extract SOs and decompose them to SFRs, as described by the meta-model in Figure 4.3. Execution principles of the MDSF Toolset are described in more detail in Section 4.4. A subset of the UML Analysis Profile stereotypes are explained in Table 4.2.

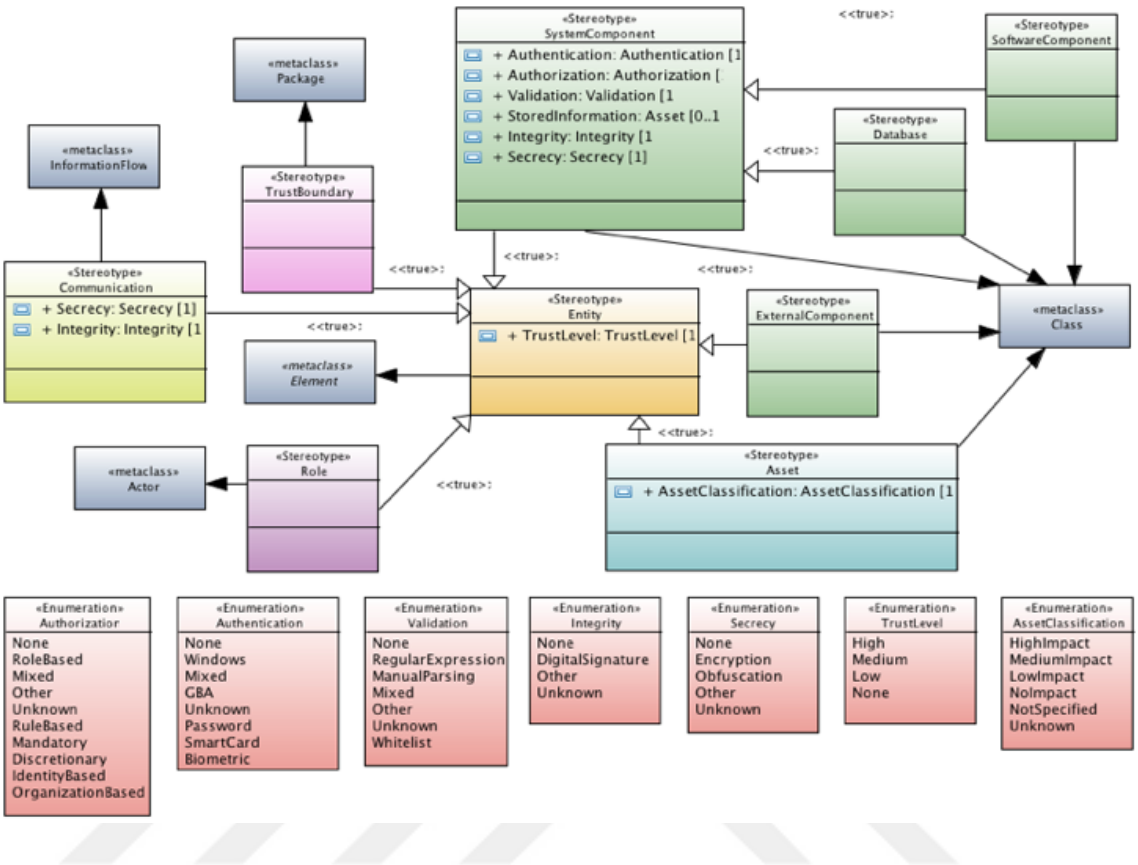


Figure 4.5: MDSF Analysis Profile (Subset)

Table 4.2: MDSF UML Analysis Profile Stereotypes (Subset)

Stereotype	**Base Class**	**Description**
Entity	Element	All other stereotypes are inherited from entity
System Component	Class	Components of the information system such as web server, database, etc.
Software Component	Class	A system component as software
Database	Class	A specialized system component to persist asset information

Continued on next page

Table 4.2 – continued from previous page

Stereotype	Base Class	Description
Asset	Class	Information which is created/read/modified during the operation of the system
TrustBoundary	Package	Trusted zone where communication between system components is assumed to be secured from external attacks
Role	Actor	Roles which interact with the system components
Communication	Information Flow	Communication between different system components

UML stereotypes are, most of the time, coupled with some tagged values. These tags enable the designer to associate additional information with the stereotyped UML entity. Table 4.3 contains descriptions for the subset of the tags associated with the stereotypes in the MDSF UML Analysis Profile. Some of the tags mentioned are of the Enumeration type.

Table 4.3: MDSF UML Analysis Profile Tags (Subset)

Tag	Owner	Description
TrustLevel [Integer(1)]	Entity	How trustable the entity is from the system point of view

Continued on next page

Table 4.3 – continued from previous page

Tag	Owner	Description
Authentication [Enumeration(1)]	System Component	Type of authentication this component has during communication with other components and actors
Authorization [Enumeration(1)]	System Component	Authorization mechanism such as role based, etc.
Validation [Enumeration(1)]	System Component	How the inputs to these components are validated such as regular expressions, parsing, etc.
StoredInformation [Asset(0..1)]	System Component	Assets persisted on this system component
DataClassification [Enumeration(1)]	Asset	Level of importance of the asset
Secrecy [Enumeration(1)]	Communication	The way the assets are transferred during a communication
Integrity [Enumeration(1)]	Communication	Method that secures the integrity of the transferred asset during the communication

All of the stereotypes are derived from a general stereotype class named Entity. This stereotype extends the UML meta class Element. Entity stereotype has one tagged value; TrustLevel. TrustLevel identifies how trustable the entity is from the system point of view. Values of TrustLevel are extendable and default values are defined in Figure 4.6.

Asset stereotype is used to define the important entities that have to be protected

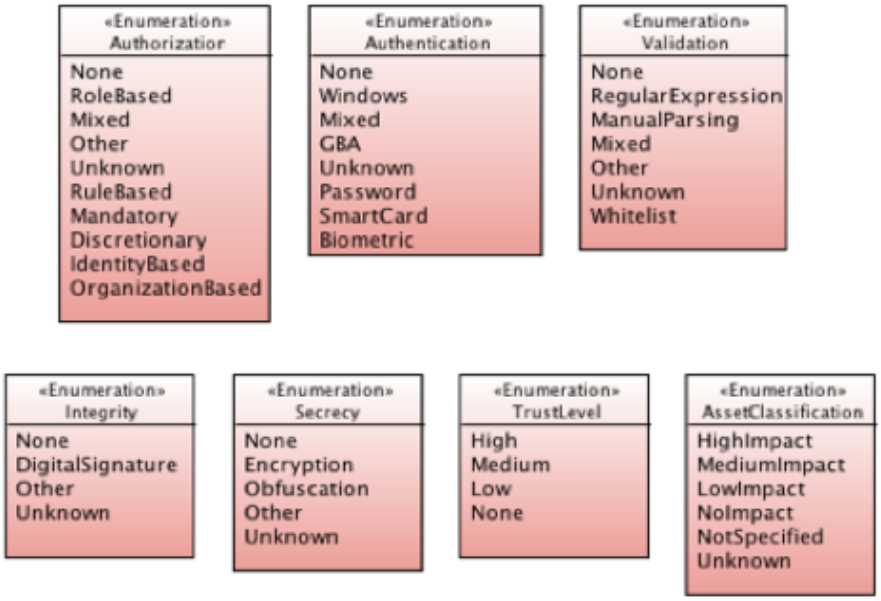


Figure 4.6: Threat Analysis Profile - Enumerations

against threats. This stereotype extends the UML meta class Class. Asset stereotype has only one additional tagged value named DataClassification. DataClassification describes the level of importance for the asset. Possible values for DataClassification can be found in Figure 4.6.

Role stereotype identifies the external actors for the use cases. This stereotype extends the UML meta class Actor. These actors may be humans or some other external systems which directly interact with the system components. Role stereotype does not have any additional tagged values.

Communication stereotype defines the interaction between different system components. This stereotype extends UML meta class InformationFlow where there's conveyed information. Communication stereotype has two additional tagged values; Secrecy and Integrity. Secrecy defines the way assets are transferred during a communication such as clear text, encrypted, etc. Integrity defines if there's any method that secures the integrity of the transferred asset during the communication. Defined values

for Secrecy and Integrity can be found in Figure 4.6.

SystemComponent stereotype is used to define separate components of the system. Components can be servers, applications, application components or even external web services. This stereotype extends UML meta class Class. SystemComponent stereotype has six additional tagged values; Authentication, Authorization and Validation, StoredInformation, Integrity and Secrecy. Authentication identifies the type of authentication this component does during communication with other components and actors. Authorization identifies the authorization mechanism such as role based, etc. Validation identifies the way inputs to these components are validated such as regular expressions, parsing, etc. Secrecy defines the way assets are stored such as clear text, encrypted, etc. Integrity defines if there's any method that secures the integrity of the stored information. Defined values for Authentication, Authorization, Validation, Secrecy and Integrity can be found in Figure 4.6.

SystemComponent stereotype is further extended to define SoftwareComponent and Database stereotypes. SoftwareComponent stereotype identifies that the component is a software application, component or web service. Database stereotype directly identifies a data store where asset information is persisted. Both of these stereotypes extend UML meta class Class.

TrustBoundary stereotype is extended from the UML meta class Package. Whenever two system components are in a trusted zone such as protected by a firewall, these system components are presented in the same package and package is stereotyped with TrustBoundary.

4.2.2. MDSF UML Design Profile

The MDSF UML Design Profile contains the stereotypes and tagged values that are used to embed security properties into the design model of the software system. A subset of the stereotypes in the MDSF UML Design Profile is given in Table 4.4.

Table 4.4: MDSF UML Design Profile Stereotypes (Subset)

Stereotype	**Base Class**	**Description**
Authentication FailureHandling	StateMachine	FIA_AFL.1 Authentication failure handling requirements
Authentication Exception	Message	Identifies authentication exception message
MaxFailure	Attribute	Marks the attribute for storing failed attempts
Authentication Operation	Operation	Identifies the operation which triggers the authentication
User	Class	Entity acting on behalf of the stakeholders
SecurityAttribute	Class Property	Security related attributes associated to an individual user
UserExtension	Class	Extension point for function merge operation
SecurityBehaviour	StateMachine	Dynamic security behavior associated with a class
OperationWith FormalBody	Operation	Operation body which contains a formal expression of the main functionality
AuditFunction	Use Case	FAU_GEN.1.1 Marks audit functions in the model
Auditable Event	Use Case	FAU_GEN.1.1 Marks auditable events in the model
Potential ViolationAnalysis	Use Case	FAU_SAA.1.1, FAU_ARP.1.1 The function which analyzes the potential violations on the audit logs

Continued on next page

Table 4.4 – continued from previous page

Stereotype	Base Class	Description
AuditLog Function	Use Case	FAU_GEN.1.1 The function which records the auditable events
LogEvent	Call Behaviour Action	FAU_SAA.1.1, FAU_ARP.1.1 Marks the log actions in the activity models of the audit functions and auditable functions
AnalyzeLog	Call Behaviour Action	FAU_SAA.1.1, FAU_ARP.1.1 Marks log analysis action in the activity model of the audit log function
AuditFunction Activity	Activity	Activity diagrams for the audit functions
AuditableEvent Activity	Activity	Activity diagrams for the auditable events such as user account creation, password change, etc.
PotentialViolation AnalysisActivity	Activity	Activity diagrams for the potential violation analysis function
AuditLog FunctionActivity	Activity	Activity diagram for the audit log function
SecureFailure	StateMachine	FPT_FLS.1 Failure with preservation of secure state requirements
TimingOf Authentication	StateMachine	FIA_UAU.1 Timing of authentication requirements
FairExchange Flow	Interaction	Enforces fair exchange flow

Continued on next page

Table 4.4 – continued from previous page

Stereotype	Base Class	Description
Trusted InformationFlow	Interaction	FDP_IFC.1, FDP_IFF.1 Enforces information flow policies
Asset	Class	Mark the assets and their trust levels
System Component	Class	Mark the system components and their trust levels
AccessControl Required	Class	FDP_ACC.1 Subset access control requirements
Role	Class	Marks the access control roles
Transaction	Operation	Marks the access controlled transactions
Segregation OfDuties	Package	Enforces segregation of duties requirements
User	Class	Users and their assigned roles
Anonymity (ongoing)	Attribute	FPR_ANO.1 Anonymity requirements

Stereotypes related to the Common Criteria SFRs are categorized according to the Common Criteria function families such as Identification and Authentication Family (FIA), Privacy Family (FPR), Protection of the TSF Family (FPT), and User Data Protection Family (FDP) in the description column of the Table 4.4. This subset of the MDSF UML Design Profile focuses mainly on the authentication, authorization, availability, information flow, audit and fair exchange security properties of software systems. Table 4.5 contains descriptions for the subset of the tags associated with the stereotypes in the MDSF UML Design Profile.

Table 4.5: MDSF UML Design Profile Tags (Subset)

Tag	**Owner**	**Description**
Authentication FailureSubState [State(1)]	Authentication Failure Handling	Identifies the state after failed authentication
Authentication SuccessState[State(1)]	Authentication Failure Handling	Identifies the state after successful authentication
AllowedBefore [State(*)]	TimingOf Authentication	Identifies the states allowed before a successful authentication
AllowedAfter [State(*)]	TimingOf Authentication	Identifies the states allowed after a successful authentication
SafeStates [State(*)]	SecureFailure	Identifies the safe states returned after an exception or failure occurs
AttributeType [Enumeration(1)]	Security Attribute	Identifies the type of security property., i.e. max login attempt or current login attempt
FormalExpression [OpaqueExpression(1)]	OperationWith FormalBody	A formal specification of function body as an opaque expression
LevelOfAudit [AuditLevels]	AuditLog Function AuditableEvent	Specifies the detail level of the audit function, i.e. minimum, basic, detailed, not specified.
EventType [AuditEventType]	LogEvent	Type of the audit event, i.e. audit function or security related function, i.e. audit function startup, audit function shutdown, security event, other, etc.

Continued on next page

Table 4.5 – continued from previous page

Tag	Owner	Description
StartClass [Class]	FairExchange Flow	Initial component for the fair exchange policy
StartReceived Asset [Class]	FairExchange Flow	Asset received by the initial component of the fair exchange policy
StopClass [Class]	FairExchange Flow	Final component of the fair exchange policy
StopReceived Asset [Class]	FairExchange Flow	Asset received by the final component of the fair exchange policy
TrustLevel [Integer]	Asset, SystemComponent	Trust levels of the system components and the assets
Transactions [Transaction(1..*)]	Role	Transactions allowed in a role
Roles [Role(0..*)]	Role	Roles under a role hierarchy
NotAllowed Transactions [Operation(*)]	Transaction	Conflicting transactions for a given transaction
Roles [Role(0..*)]	User	Assigned roles for a user

The UML model for the stereotypes related to Identification and Authentication family are given in Figure 4.7

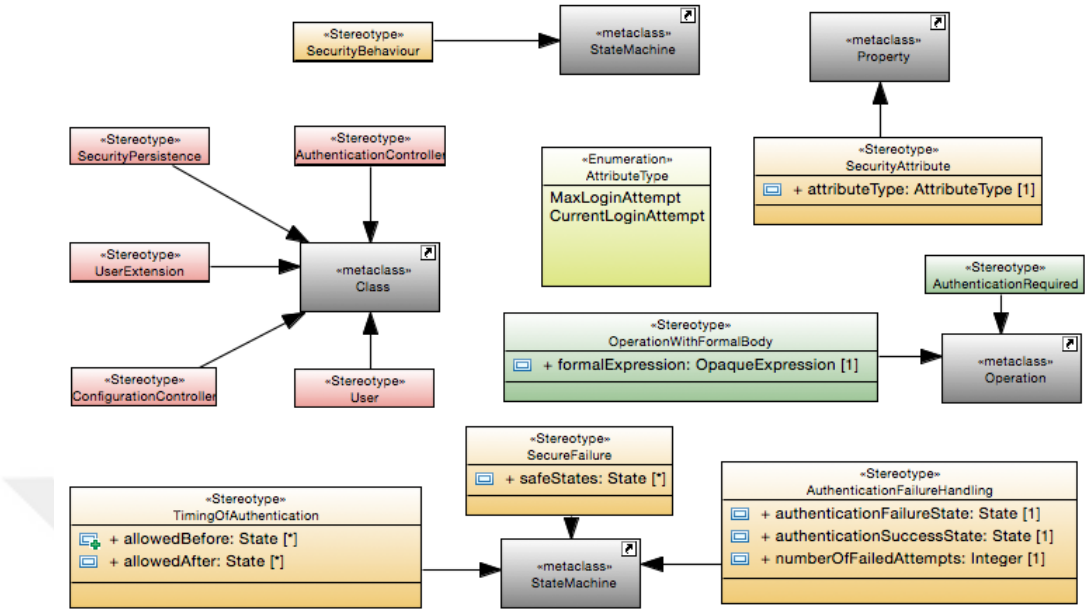


Figure 4.7: MDSF Design Profile: Identification and Authentication

"User" stereotype is used to mark a design class which represents a real world user. During security design process this "User" class will be responsible in several aspects such as authentication, authorization, access control, audit, etc.

"SecurityAttribute" stereotype is used to mark a design class, which is used to hold a security, related attribute associated to individual users. "attributeType" tagged value is used to identify the type of security property.

"UserExtension" stereotype is used to mark a design class, which is a generalization of a User class defined in the aspect library. This extension represents a connection point where the identified aspect is going to be merged into the design model.

"ConfigurationController" stereotype is used to mark a design class, which handles administrative tasks such as managing values of security properties.

"AuthenticationController" stereotype is used to mark a design class, which handles authentication requests, received from the users and clients.

"SecurityBehaviour" stereotype is used to mark a state machine, which describes dynamic behavior of a security operation such as login.

"OperationWithFormalBody" stereotype is used whenever there is a need to formally model check operation body. With this stereotype it's expected that a formal specification of function body is provided as an opaque expression in the formalExpression tagged value. Whenever an operation is annotated with this stereotype, model checker can be used to model check pre and post OCL constraints of the operation on the formal definition.

"TimingOfAuthentication" stereotype is used to mark which states are allowed for transition before and after successful authentication.

"AuthenticationFailureHandling" stereotype is used to mark which state will be active after a failure or authentication success. Also number of failed attempts can be defined within this stereotype.

"SecureFailure" stereotype is used to mark which states are secure after a failed authentication.

The UML model for the stereotypes related to Access Control family are given in Figure 4.8

"Role" stereotype is used to define the access control roles within the system.

"User" stereotype is used to assign roles to users in the system.

"Transaction" stereotype is used to mark the access controlled operations and also conflicting operations in terms of segregation of duties requirements.

The UML model for the stereotypes related to Audit family are given in Figure 4.9

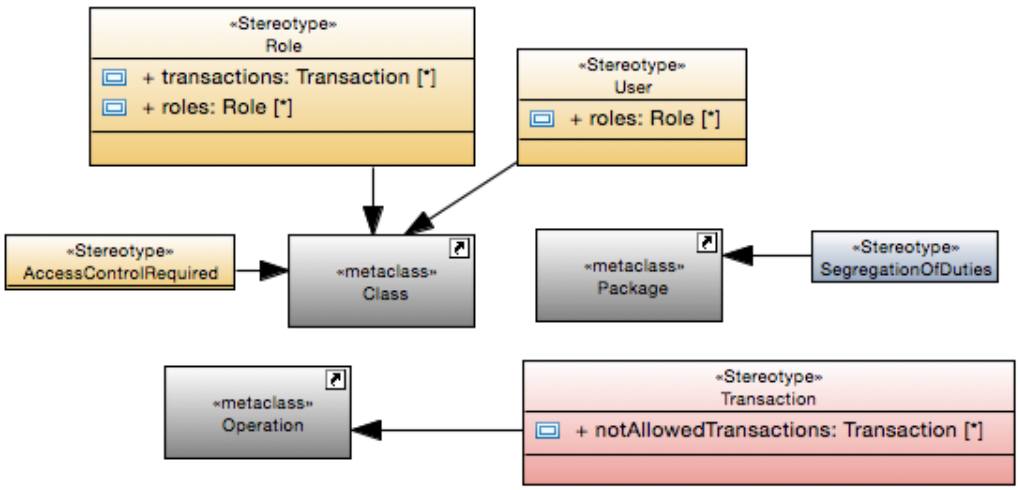


Figure 4.8: MDSF Design Profile: Access Control

"AuditLogFunction" stereotype is used mark the audit log function use case in the model. Also the audit log level can be defined with this stereotype.

"AuditableEvent" stereotype is used to mark the security related function use cases in the model. Examples are registration, account creation, account deletion, login, failed login, etc. Also the audit log level per event type can be set using the same stereotype. If the auditableEvent log level is higher or equal to the log level of the AuditLogFunction, then the event will be logged.

"AuditFunction" stereotype is used to mark the audit use cases such as verify audit logs, audit log configuration, export audit log, etc.

"PotentialViolationAnalysis" stereotype is used to mark the violation analysis use case.

"AuditLogFunctionActivity" stereotype is used to mark the activity diagram of the audit log function.

"AuditableEventActivity" stereotype is used to mark the activity diagram of the auditable event function.

"AuditFunctionActivity" stereotype is used to mark the activity diagram of the audit function.

"PotentialViolationAnalysisActivity" stereotype is used to mark the potential violation analysis function.

"LogEvent" stereotype is used to mark the log event operation call within the auditable event activity diagram.

"AnalyzeLog" stereotype is used to mark the analyze log event operation call within the audit log function activity diagram

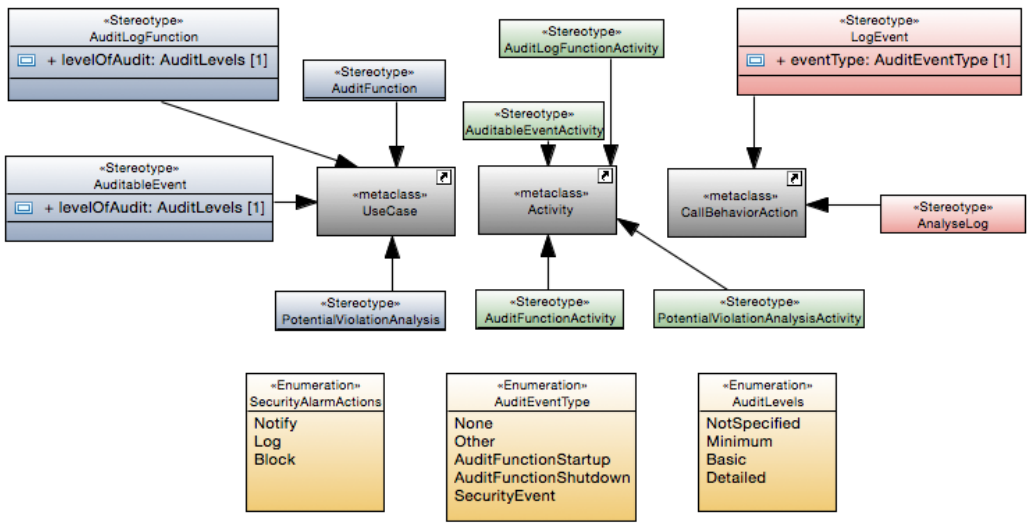


Figure 4.9: MDSF Design Profile: Audit

The UML model for the stereotypes related to Information Flow family are given in Figure 4.10

"TrustedInformationFlow" stereotype is used to enforce information flow control policies on the interaction diagrams of the model.

"FairExchangeFlow" stereotype is used to enforce fair exchange flow control policies on the interaction diagrams of the model.

"SystemComponent" stereotype is used to mark the system components and assign them trust levels to be used in the trusted information flow control policy.

"Asset" stereotype is used to mark the conveyed information in the interaction diagrams and assign them trust levels to be used in the trusted information flow control policy.

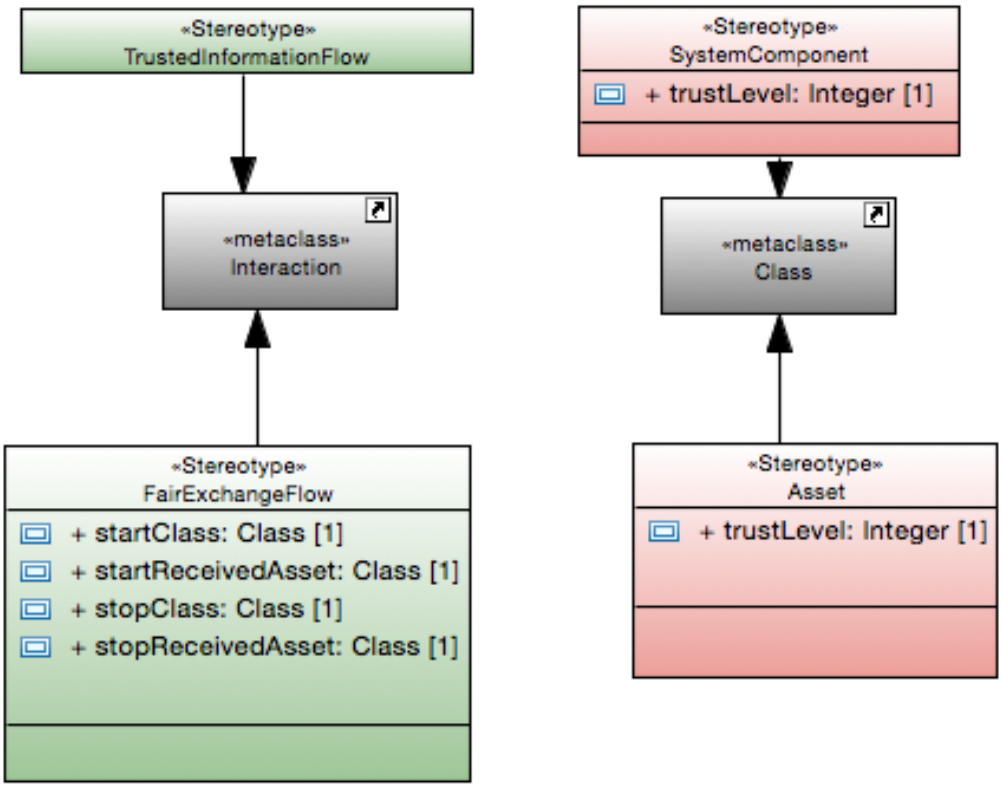


Figure 4.10: MDSF Design Profile: Information Flow

Detailed examples of stereotypes and tags in the MDSF UML Analysis Profile and Design Profile are presented in association with the proof of concept case study in Section 5.

4.3. MDSF Library

4.3.1. Usage of OCL in MDSF

MDSF can work with any UML design tool that supports external UML profiles and can export the model as an Extensible Markup Language (XML) Meta-data Interchange (XMI) document. We used Eclipse Model Development Tools (MDT) as the UML design environment and extended it to create a UML application development environment. The MDT implementations of UML and OCL are used for this purpose. The four tools of MDSF are developed on top of this development environment and are available for future customizations and developments. In the MDSF Toolset, OCL is used heavily for verification and validation of UML analysis and design diagrams and to check consistencies between models. As an example, at every step of the execution, the MDSF Toolset checks the consistency and ensures that the models are well formed using pre-defined OCL statements. Example OCL in Table 4.6 checks whether all of the classes in the analysis model have correctly applied stereotypes (SoftwareComponent, Database and Asset).

Table 4.6: Example OCL Invariant for Checking UML Stereotypes

```
self.allOwnedElements() ->
    select ( a | a.oclIsTypeOf( uml::Class ) ) ->
    forAll ( b | b.getAppliedStereotypes().name ->
        intersection( Set (
            'SoftwareComponent'
            'Database',
            'Asset' ) ) -> size() > 0 )
```

Another example OCL invariant for checking the consistency of the analysis model is given in Table 4.7. This example OCL invariant checks whether all of the communication methods are defined among different solution components. In addition, the

example invariant also checks whether the type of information transferred between components is defined for each communication line.

Table 4.7: Example OCL Invariant for Model Correctness

```
self.allOwnedElements() ->
select( a | a.oclIsTypeOf(uml::Association ) ) ->
forAll( b | b.getAppliedStereotypes().name ->
    includes( 'Communication' ) implies
    b.oclAsType( uml::Association ).memberEnd ->
    select( p | p.getAppliedStereotypes().name ->
        includes( 'ConveyedInformation' ) ) ->
        size() = 1
    and
    b.oclAsType( uml::Association ).memberEnd ->
    select( p | p.getAppliedStereotypes().name ->
        includes( 'ConveyedInformation' ) ) ->
        first().oclAsType( Property ).
        extensionConveyedInformation.
        Information.
        oclIsTypeOf( ThreatAnalysis::Asset )
)
```

4.3.1.1. OCL for Threat Identification. Framework contains all OCL statements and invariants necessary to identify possible threats present in the threat analysis model. These OCL constraints, when executed on a model stereotyped with the Threat Analysis Package, show if a specific threat exists for the model under investigation.

Table 4.8: Example OCL Invariant for Information Disclosure Threat Identification

```
package uml context Package
inv: self.allOwnedElements() ->
    select (a | a.oclIsTypeOf(uml::Association)) ->
    select (a | a.getAppliedStereotypes().name->
        includes('Communication')) ->
    forAll (a | a.oclAsType(Association).memberEnd ->
        select (p | p.getAppliedStereotypes().name ->
            includes ('ConveyedInformation')) ->
        first().type.oclAsType (Class).
        extension_SystemComponent.TrustLevel >
        a.oclAsType (Association).memberEnd ->
        select (p | not p.getAppliedStereotypes().name ->
            includes ('ConveyedInformation')) ->
        first().type.oclAsType (Class).
        extension_SystemComponent.TrustLevel
        implies a.oclAsType (Association).memberEnd ->
        select (p | p.getAppliedStereotypes().name ->
            includes ('ConveyedInformation')) ->
        first().oclAsType (Property).extension_ConveyedInformation.
        Information.oclAsType (ThreatAnalysis::Asset).
        DataClassification =
        ThreatAnalysis::AssetClassification::NoImpact)
Endpackage
```

OCL constraint in Table 4.8 selects all associations which has “Communication” stereotype applied and checks if the TrustLevel tagged value of source end is smaller than the TrustLevel tagged value of the destination end. In other words, if receiver of critical information is less trusted then the sender of the information then there’s a possibility that information disclosure may occur.

Table 4.9: Example OCL Invariant for Spoofing Threat Identification

```
package uml context Package
inv: self.allOwnedElements() ->
select (a | a.oclIsTypeOf (uml::Association)) ->
select (a | a.getAppliedStereotypes().name->
   includes ('Communication')) ->
forAll (a | a.oclAsType(Association).
   extension_Communication.TrustLevel >
   a.oclAsType (Association).memberEnd ->
   select (p | p.getAppliedStereotypes().name->
      includes ('ConveyedInformation')) ->
   first().type.oclAsType (Class).extension_SystemComponent.
   TrustLevel implies a.oclAsType (Association).memberEnd ->
   select (p | p.getAppliedStereotypes().name ->
       includes('ConveyedInformation')) ->
   first().oclAsType(Property).
   extension_ConveyedInformation.Information.oclAsType
   (ThreatAnalysis::Asset).DataClassification =
   ThreatAnalysis::AssetClassification::NoImpact)
Endpackage
```

OCL constraint in Table 4.9 selects all associations, which has “Communication” stereotype is applied and checks if the TrustLevel tagged value of the communication line is bigger then the TrustLevel tagged value of the source end. In other words, if sender of critical information is less trusted than the communication line then there’s a possibility that spoofing may occur.

OCL constraint in Table 4.10 checks all the information flows in the system model and identifies information disclosure threat if an important asset is conveyed in an information flow and either that information flow is not encrypted or the target and the source of the information flow is not in the same trusted zone.

OCL constraint in Table 4.11 checks all the information flows in the system model and identifies information disclosure threat if a system component receives conveyed information from an external component and it does not check the validity of the input using white lists, regular expressions or parsing.

Table 4.10: Example OCL Invariant for Info. Disclosure Threat Identification - 2

```
package uml
context uml::Package
inv constraint1 : self.allOwnedElements()->
select(a | a.oclIsTypeOf(uml::InformationFlow)).
oclAsType(uml::InformationFlow)->
forAll(b | b.conveyed.getValue(
    b.conveyed.getAppliedStereotype('profile::Asset')->
    asOrderedSet()->first(),'AssetClassification').
    oclAsType(uml::EnumerationLiteral).name->
    intersection(
        Set'HighImpact','MediumImpact','LowImpact','NotSpecified','Unknown'
        )->size() <> 0
    implies
    (b.getValue(
    b.getAppliedStereotype('profile::Communication')->
    asOrderedSet()->first(),'Secrecy').
    oclAsType(uml::EnumerationLiteral).name->
    includes('Encryption') or
    ((b.informationSource.owner = b.informationTarget.owner) and
    b.informationSource.owner.
    getAppliedStereotype('profile::TrustBoundary')->
    asOrderedSet()->size()<>0))
)
endpackage
```

Table 4.11: Example OCL Invariant for Info. Disclosure Threat Identification - 3

```
context uml::Package
inv constraint1 : self.allOwnedElements()->
select(oclIsTypeOf(uml::Class)).oclAsType(uml::Class)->
forAll(a | a.getAppliedStereotype('profile::SoftwareComponent')->
asOrderedSet()->size() <> 0 and
a.name->intersection(
   self.allOwnedElements()-> select(b |
      b.oclIsTypeOf(uml::InformationFlow)).
      oclAsType(uml::InformationFlow)->select (c |
         c.informationSource.name->intersection(
         self.allOwnedElements()-> select(d |
         d.oclIsTypeOf(uml::InformationFlow))
         .oclAsType(uml::InformationFlow).
         informationSource->asSet()->select(e |
         e.getAppliedStereotype('profile::ExternalComponent')->
         asOrderedSet()->size() <> 0)
         .oclAsType(uml::Class).name
   )->size()<>0).informationTarget.name)->size() <> 0
implies a.getValue(
a.getAppliedStereotype('profile::SoftwareComponent')->
asOrderedSet()->first(),'Validation').
oclAsType(uml::EnumerationLiteral).name->intersection(
Set'RegularExpression','ManualParsing','Mixed','Other','Whitelist')->
size() <> 0)
```

4.3.1.2. OCL for Model Validity Check. OCL is used to check if MDSF UML Profiles are correctly applied to the UML model. Below constraints are automatically checked when the UML model is first input to the Security Requirement Extraction Tool:

Table 4.12: Example OCL Invariant for Model Validity Check

```
inv: self.allOwnedElements() ->
select (a | a.oclIsTypeOf(uml::Class)) ->
forAll (b | b.getAppliedStereotypes().name ->
   intersection (Set'SoftwareComponent','Database','Asset') ->
   size() > 0
)
```

OCL invariant in Table 4.12 checks if all classes are marked as either "Software-Component", "Database" or "Asset".

Table 4.13: Another Example of an OCL Invariant for Model Validity Check

```
inv: self.allOwnedElements()->
select(a | a.oclIsTypeOf(uml::Association)) ->
forAll (b | b.getAppliedStereotypes().name ->
    includes ('Communication') implies
    b.oclAsType (uml::Association).memberEnd ->
    select (p | p.getAppliedStereotypes().name ->
        includes ('ConveyedInformation')) ->
    size()=1and b.oclAsType(uml::Association).memberEnd ->
    select (p | p.getAppliedStereotypes().name ->
        includes ('ConveyedInformation')) ->
     first().oclAsType (Property).
    extension_ConveyedInformation.Information.
    oclIsTypeOf (ThreatAnalysis::Asset)
)
```

OCL invariant in Table 4.13 gets every association which has "Communication" stereotype applied and checks if one and only one "memberEnd" of this association has "ConveyedInformation" stereotype applied and this member end has an "Asset" class assigned for the "Information" tagged value.

Similar validity constraints are also used during security aspect weaving procedures.

4.3.1.3. OCL for Definitions. OCL 2.0 allows definition of functions and usage of these functions in advanced invariants and queries. MDSF Library contains necessary definitions which helps writing advanced constraints on the analysis and design models.

Some of these definitions are presented below:

Table 4.14: OCL for Stereotype Inquiry

```
def:
numberOfClassesOfType(s : String) : Integer =
self.allOwnedElements()->select(a | a.oclIsTypeOf(uml::Class))->
select(b | b.getAppliedStereotypes().name->includes(s))->size()
```

OCL function in Table 4.14 gets the name of a stereotype and returns the number of classes which has that stereotype applied in the model.

Table 4.15: OCL for TrustLevel Check

```
def:
isTrustLevel(className : String, trustLevel: Integer) : Boolean =
self.allOwnedElements()->
select(a | a.oclIsTypeOf(uml::Class))->
select(b | b.getAppliedStereotypes().name->
    includes('SoftwareComponent'))->
select(c | c.oclAsType(Class).name = className)->
forAll(d | d.oclAsType(Class).
extension_SoftwareComponent.TrustLevel = trustLevel)
```

OCL function in Table 4.15 gets the name of a class and an Integer TrustLevel value, and checks if that class has "SoftwareComponent" stereotype applied and has Trust Level value equal to the TrustLevel value, which is input to the function.

Table 4.16: OCL for TrustLevel Inquiry

```
def:
getTrustLevel(className : String) : Integer =
self.allOwnedElements()->
select(a | a.oclIsTypeOf(uml::Class))->
select(b | b.getAppliedStereotypes().name->
    includes('SoftwareComponent'))->
any(c | c.oclAsType(Class).name = className).
oclAsType(Class).extension_SoftwareComponent.TrustLevel
```

OCL function in Table 4.16 gets the name of a class and returns the associated TrustLevel tagged value.

Table 4.17: OCL for Class Inquiry

```
def:
getClassesWithStereotype(stereotypeName: String): Bag(Class) =
self.allOwnedElements()->
select(a | a.oclIsTypeOf(uml::Class))->
select(b | b.getAppliedStereotypes().name->
    includes(stereotypeName))->
    collect(oclAsType(Class)
)
```

OCL function in Table 4.17 gets the name of a stereotype and returns the list of classes which has that stereotype applied in the model.

Table 4.18: OCL for Max Login Attempt Check

```
def:
hasMaxLoginAttemptProperty() : Boolean =
self.allOwnedElements()->
select(a | a.oclIsTypeOf(uml::Class))->
collect(oclAsType(uml::Class).attribute)->
select(b | b.getAppliedStereotypes().name->
   includes('SecurityAttribute'))->
select(c | c.oclAsType(uml::Property).
   extension_SecurityAttribute.attributeType =
   IdentificationAndAuthentication::AttributeType::MaxLoginAttempt
)->size() = 1
```

OCL function in Table 4.18 checks if model has any security attributes to manage max allowed number of unsuccessful login attempts.

Table 4.19: OCL for Current Login Attempt Inquiry

```
def:
hasCurrentLoginAttemptProperty() : Boolean =
self.allOwnedElements()->
select(a | a.oclIsTypeOf(uml::Class))->
collect(oclAsType(uml::Class).attribute)->
select(b | b.getAppliedStereotypes().name->
    includes('SecurityAttribute'))->
select(c | c.oclAsType(uml::Property).
    extension_SecurityAttribute.attributeType =
    IdentificationAndAuthentication::AttributeType::CurrentLoginAttempt
)->size() = 1
```

OCL function in Table 4.19 checks if model has any security attributes to manage current number of unsuccessful login attempts.

Table 4.20: OCL for Security Attribute Inquiry

```
def:
numberOfAttributesOfType(s : String) : Integer =
self.allOwnedElements()->
select(a | a.oclIsTypeOf(uml::Property))->
select(b | b.getAppliedStereotypes().name->
    includes(s)
)->size()
```

OCL function in Table 4.20 returns the number of attributes in the model which are identified as security attributes.

Table 4.21: OCL for Max Login Attempt Inquiry

```
def:
getMaxLoginAttemptDefault() : String =
self.allOwnedElements()->
select(a | a.oclIsTypeOf(uml::Class))->
collect(oclAsType(uml::Class).attribute)->
select(b | b.getAppliedStereotypes().name->
    includes('SecurityAttribute'))->
select(c | c.oclAsType(uml::Property).
    extension_SecurityAttribute.attributeType =
    IdentificationAndAuthentication::AttributeType::MaxLoginAttempt
)->asOrderedSet()->first().oclAsType(uml::Property).default
```

OCL function in Table 4.21 returns the default value of MaxLoginAttempt property in the model.

4.3.1.4. OCL for Security Requirements Enforcement. OCL can also used to enforce some security requirements on the design model. OCL specification allows designers to write post constraints on methods of classes. Usage of post constraints can best be described by a use case:

Figure 4.11: Security Requirements Enforcement with OCL Class Diagram for Use Case

Figure 4.11 contains a UML Class diagram describing a PasswordManager class. In the use case, this class is responsible to provide operations to generate passwords and check user generated passwords according to the defined security policy. Security

Policy details can easily be provide with body and post conditions on the UML class diagram.

context PasswordManager::checkPassword(): Boolean
body: (password.size()>6 and password.size()<8)

Above OCL body condition puts a requirement on the password checking operation and restricts the length of the input password between six and eight characters.

context PasswordManager::createPassword(password: String): void
post: (result.size()>6 and result.size()<8)

Above OCL post condition sets a requirement on generated passwords and limits the length of generated password between six and eight characters.

This type of constraints is useful to enforce some basic security constraints on the design model. Existence of such constraints can validate that the design actually enforces some mandatory security functional requirements such as FIA_SOS.1.1 and FIA_SOS.2.1.

FIA_SOS.1.1 states that the TSF shall provide a mechanism to verify that secrets meet a defined quality metric. Security analysts need to define the quality metric for the secrets used within the system. In above use case the metric is only defined on the length of the passwords and enforced using the OCL body condition.

FIA_SOS.2.1 states that the TSF shall provide a mechanism to generate secrets that meet a defined quality metric. Above OCL post condition checks if the generated password actually meets the defined quality metric. However it does not actually enforce the requirement on the password generation mechanism. In order to check if password generation mechanism always generates passwords that meet the defined quality metrics, some additional formal methods need to be used.

FAU_GEN.1.1 states that the TSF shall be able to generate an audit record of the following auditable events:

- Start-up and shutdown of the audit functions;
- All auditable events for the [selection, choose one of: minimum, basic, detailed, not specified] level of audit; and
- Other specifically defined auditable events.

OCL function in Table 4.22 checks if all audit functions are logged:

Table 4.22: OCL for Audit Log Requirement

```
inv isAuditFunctionsLogged : UseCase.allInstances()->
select(getAppliedStereotypes().name->
   includes('AuditFunction'))->
forAll(auditFunction | auditFunction.oclAsType(uml::UseCase).
   include.target.oclAsType(uml::UseCase).
   getAppliedStereotypes().name->
   includes('AuditLogFunction'))
```

OCL function in Table 4.23 checks if audit functions records start-up and shutdown events.

Table 4.23: OCL for Audit Log Requirement - 2

```
inv doAuditFunctionsRecordStartupAndShutdown :
Activity.allInstances()->
select (a | a.getAppliedStereotypes().name->
   includes('AuditFunctionActivity'))->
forAll(b | b.allOwnedElements()->
   select(c | c.oclIsKindOf(uml::CallBehaviorAction))->
   select(d | d.getAppliedStereotypes().name->
      includes('LogEvent'))->size() = 2 and
   b.allOwnedElements()->
   select(c | c.oclIsKindOf(uml::CallBehaviorAction))->
   select(d | d.getAppliedStereotypes().name->
      includes('LogEvent'))->
   collect(e | e.getValue(e.getAppliedStereotypes()->
      select(name = 'LogEvent')->asOrderedSet()->
      first(),'eventType').oclAsType(uml::EnumerationLiteral).name)->
   includesAll(Bag'AuditFunctionStartup','AuditFunctionShutdown'))
```

4.3.2. Security Aspects

In order to reduce human errors during software lifecycle (design, development and testing) we propose a library, which contains appropriate designs for several security functions which correspond to different SFRs. Some of the SFRs will be matched to the appropriate security aspects from the library and with the automated tools and processes these aspects will be weaved into the TSF design model. Jurjens, in his book [19], presented an idea to apply security patterns into UML designs. In this study we take this idea, extend it to aspect oriented approach and propose a realization for embedding security aspects into UML designs of software:

- U: Base UML diagram, which does not contain any security requirements
- α: Manual function to apply security requirements into the design
- β: Semi-automated function to select and merge aspects into the design

- U'=$\beta(\alpha(U))$

In this study, we propose a semi-automated function, which is generic enough to be applied to a wide variety of information systems. Function is semi automatic because for some SFRs there may be several applicable security aspects and selection must be made with human interaction. This may also be handled fully automated by implementing a mechanism, which take some additional factors such as performance, cost, etc. into account and select the most appropriate security aspect automatically.

There are 11 security functional families in the Common Criteria:

- Security Audit
- Communication
- Cryptographic Support
- User Data Protection
- Identification and Authentication
- Security Management
- Privacy
- Protection of the TSF
- Resource Utilization
- TOE Access
- Trusted Path/Channels

In the coming sections, some examples of security functional components and security functional requirements within these families are taken into consideration and security aspects which provide security solution designs satisfying these security functional requirements are explained.

Both static and dynamic UML diagrams and UML profiles are utilized in order to create solution designs (security aspects), which fulfill these security functional requirements. There are two types of validation constraints for each security aspect:

- Pre-validation constraints: These constraints are checked on the destination design model before the security aspect is weaved into. If validation check fails, aspect weaving does not continue.
- Post-validation constraints: These constraints are checked after security aspect weaving is completed. If these constraints fail design model is not updated and changes are rolled back.

Constraints are stored in the persistence layer together with security aspect information. Every constraint is marked as mandatory or optional. If a constraint is marked as mandatory and it fails, execution of aspect weaving stops and changes are rolled back. If the constraint is marked as optional, execution continues however a warning message is displayed to designer.

For UML design diagrams and OCL validation constraints, MDSF UML profiles are used whenever required.

4.3.2.1. Security Aspect Design for FIA_ATD.1.1. FIA_ATD.1.1 states that a list of security attributes shall be maintained for individual users. These security attributes may include a wide range of parameters such as user identifier, security classification, and clearance level. This is why a generic design where every user can have some specific security attributes (listed in the requirements) and dynamic attributes is created as shown in Figure 4.12.

A simple example where this security aspect is weaved into a design model is explained in the proof of concept case study in Section 5.

Below, a list of constraints associated with this security aspect is given.

Mandatory Pre Validation Constraints

Mandatory pre validation constraints check if security aspect was previously ap-

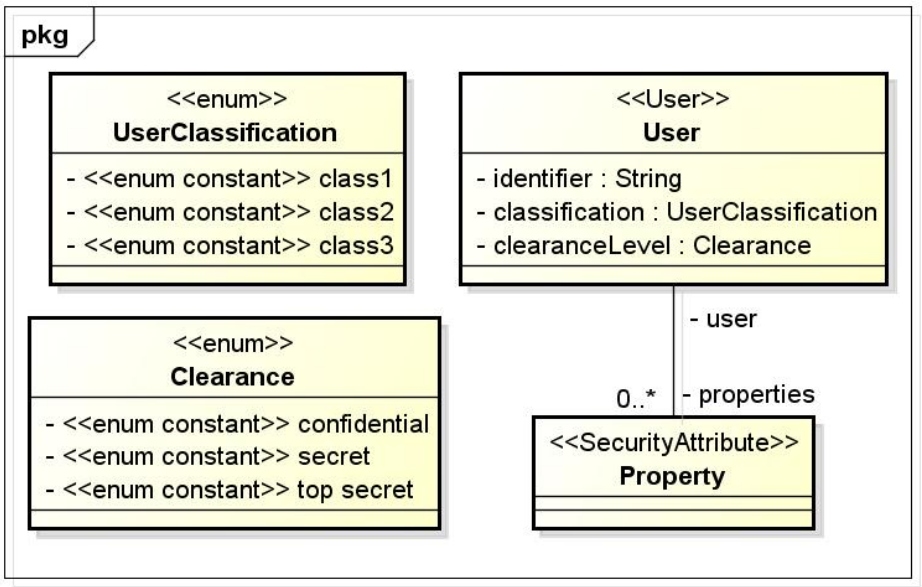


Figure 4.12: FIA_ATD.1.1 User Attribute Definition

plied into the design model.

inv: numberOfClassesOfType('User') = 0
inv: numberOfClassesOfType('SecurityAttribute') = 0

Optional Pre Validation Constraints

Optional pre validation constraint checks if any extension point for this aspect is defined in the design model. If extension point is not defined it's not possible to bind aspect and model elements using generalization associations. However in this case, execution continues.

inv: numberOfClassesOfType('UserExtension') > 1

Mandatory Post Validation Constraints

inv: numberOfClassesOfType('User') = 1
inv: numberOfClassesOfType('SecurityAttribute') = 1

inv: numberOfClassesOfType('UserExtension') = 0

Optional Post Validation Constraints

No optional post constraints are defined for this security aspect.

4.3.2.2. Security Aspect Design for FIA_AFL.1.1 and FIA_AFL.1.2. FIA_AFL.1.1 states that system shall have control over number of unsuccessful authentication attempts. In the security aspect design, this security functional requirement is interpreted as below:

req1 System shall maintain a parameter to store allowed maximum number of unsuccessful authentications

req2 Allowed maximum number of unsuccessful authentications default value shall be set to 3

req3 Whenever a user exceeds the maximum number of unsuccessful authentications, system shall detect it and lock the user account.

In order to create a design, which will satisfy above requirements, this security aspect contains a static UML class diagram together with a state machine which specifies dynamic behavior. Since requirements contain both static and dynamic properties, verification consists of both OCL invariant execution and model checking the state space with PROMELA and SPIN.

Figure 4.13 contains the class diagram and for the security aspect:

There are two main classes in the aspect design class diagram. LoginManager class is the authentication controller, which is responsible to receive authentication requests and manage user accounts. UserAccount is the class, which is associated to user entity and handles actual authentication functionality.

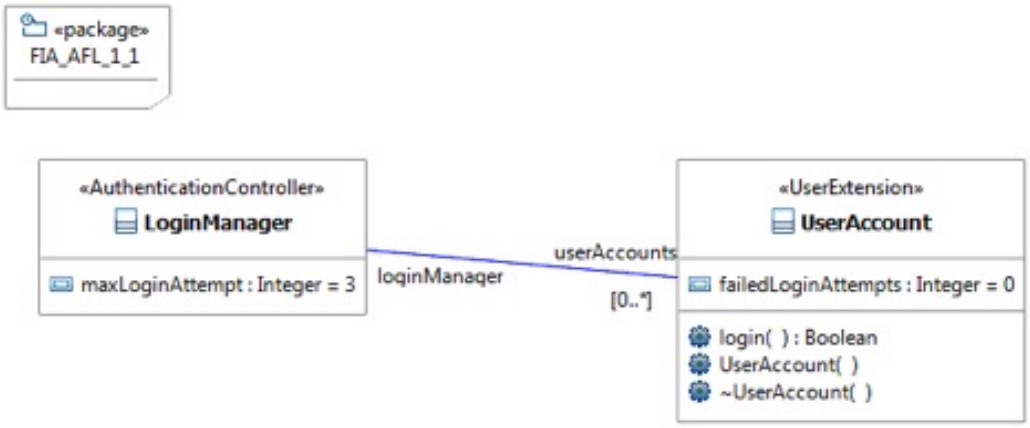


Figure 4.13: FIA_AFL, Authentication Failures

There are two security parameters marked with “SecurityParameter” stereotype in the above diagram. “SecurityParameter” stereotype is not displayed in the diagram to have a clearer picture. Security parameters are “maxLoginAttempt” and “failedLoginAttempts”. “maxLoginAttempt” holds the maximum allowed number of unsuccessful authentication attempts value. This parameter has 3 as default value. “failedLoginAttempts” holds the number of unsuccessful authentication attempts for a specific user account at a point in time. Initial value of this parameter is 0.

In order to provide a design, which satisfy dynamic requirements as well as static ones, additional UML diagrams such as state machine diagrams, sequence diagrams and activity diagrams has to be used. Below state machine diagram (Figure 4.14) belongs to “UserAccount” class and describe the behavior of the class based on external and internal triggers such as operation calls and signal events.

When a new UserAccount is created, user account object enters into “Initial” state. On “Initial” state, if a “login” attempt is done, target state changes based on the result of the authentication function. If authentication operation is successful, object’s state changes to “LoggedIn”. If authentication operation is not successful, “failedLoginAttempts” property of “UserAccount” is incremented by one. If number of “failedLoginAttempts” is still less then “maxLoginAttempt” property of the “Login-

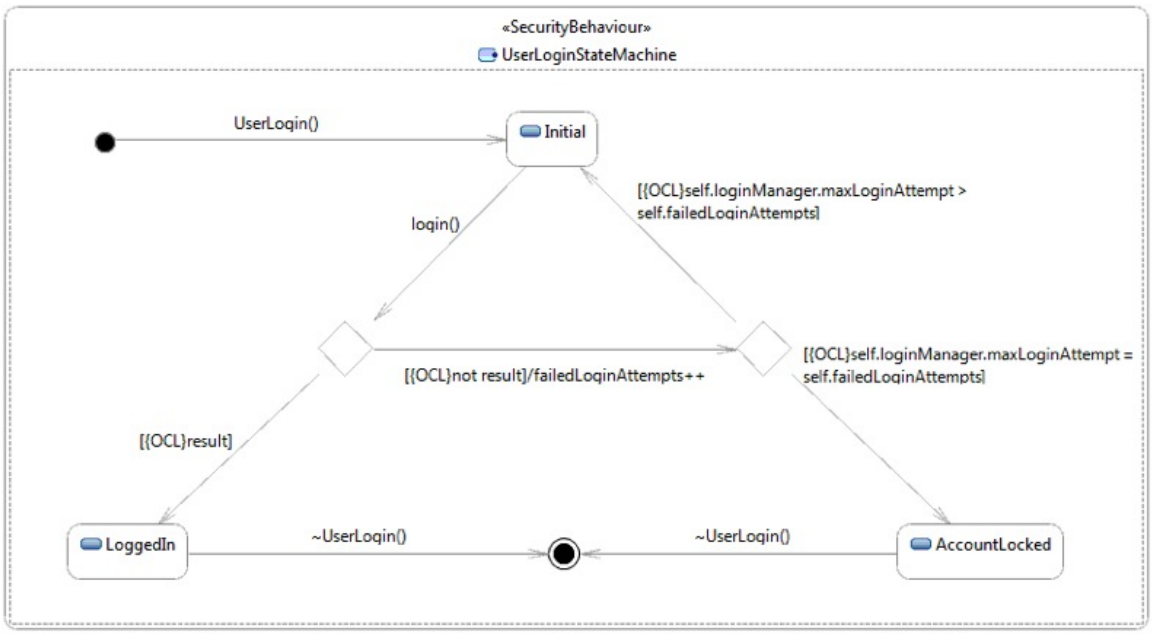


Figure 4.14: Dynamic Behavior for FIA_AFL

Manager" objects, target state is "Initial". If number of "failedLoginAttempts" is equal to "maxLoginAttempt" property, target state is "AccountLocked".

In order to check static requirements on the aspect design model, below OCL statements are created. Please refer to previous sections for details of OCL definitions used in the constraints.

Mandatory Pre Validation Constraints

Mandatory pre validation constraints check if security aspect was previously applied into the design model.

inv: numberOfClassesOfType('UserExtension') = 0

inv: numberOfClassesOfType('AuthenticationController') = 0

inv: not hasCurrentLoginAttemptProperty()

inv: not hasMaxLoginAttemptProperty()

Optional Pre Validation Constraints

No optional pre constraints are defined for this security aspect.

Mandatory Post Validation Constraints

inv: numberOfClassesOfType('AuthenticationController') = 1
inv: numberOfClassesOfType('UserExtension') = 1
inv: numberOfAttributesOfType('SecurityProperty') = 2
inv: hasCurrentLoginAttemptProperty() //satisfies [req1]
inv: hasMaxLoginAttemptProperty() //satisfies [req1]
inv: getMaxLoginAttemptDefault() = '3' //satisfies [req2]

Optional Post Validation Constraints

No optional post constraints are defined for this security aspect.

[req1] and [req2] can be both checked with execution of above OCL invariants on the design model. However it's not possible to verify [req3] using OCL execution on the static design model. This is why, state machine of Figure 4.14 is converted to PROMELA and model checked in SPIN tool. During model checking, some LTL properties are verified on the model to verify [req3]:

[] ((failedLoginAttempts==3) -> (state==ACCOUNTLOCKED || state==FINAL))

Based on the results of model checking, security aspect design satisfies [req3]:

State-vector 36 byte, depth reached 135, errors: 0
179 states, stored
81 states, matched
260 transitions (= stored+matched)
402 atomic steps

hash conflicts: 0 (resolved)

Model checking can also be applied to a design model where security aspects are not used and design is created manually. Details of UML model transformation to PROMELA and model checking are explained in Section 4.4.

A proof of concept case study to explain details of model checking is provided in Section 5

4.3.3. Threat Library

Saeki et. al. used the concept of a threat library in [54]. The Common Criteria does not specify a common set of possible threats and security objectives. This is why they used Extended Commercially Oriented Functionality Class (E-COFC, ECMA-271) [60]. It's basically a protection profile of the Common Criteria, which contains a set of threats, a set of security objectives, the relation between the threats and security objectives and the relation between the objectives and security functional components. This threat library is selected for security analysis and is open to future expansion. Threat library is strongly related to the MDSF UML Profile, Threat Analysis Package. This UML Profile Package contains necessary OCL constructs, when validated, identifies the existence of possible threads. The list of possible threats is presented in Table A.1.

4.3.4. Security Objective and SFR Decomposition Library

Threats will be converted to the security objectives and security functional components as described in E-COFC [60]. Library contains a mapping of every security functional component to the associated SFRs. Security analyst is responsible to select appropriate SFRs from the library. List of security objective definitions is given in Table A.2. Mapping of security objectives to threats is given in Table A.3.

4.4. MDSF Toolset

MDSF provides four tools, assisted by MDSF Library, to support the analysis, design and verification processes. The four tools (Threat Identification Tool (TIT), Security Requirement Extraction Tool (SRET), Security Function Merger Tool (SFMT) and PROMELA Generator Tool (PGT)) and their uses are described below. Figure 4.15 shows the MDSF Toolset usage flow in seven steps. Third party tools such as Spin Model Checker used in the MDSF Toolset are shown as boxes with dark colors. We grouped the MDSF Toolset usage steps based on the software development phases (analysis, design and verification).

The following sections explain how the MDSF Toolset is used at each phase.

4.4.1. Toolset Overview

4.4.1.1. Analysis Steps. In step 1, analysts prepare an analysis model using any UML design tool. The analysis model must contain the UML diagrams below to be used in step 2:

- Class Diagram: Standard class diagram created for the analysis model.
- Critical Asset Diagram: Critical assets in the class diagram are marked with the "Asset" stereotype and the associated tagged values from the MDSF UML Analysis Profile.
- Use Case Descriptions: Describes the flow of information between system components for the selected use cases. The flow of information is shown using the UML sequence diagrams annotated with stereotypes (ConveyedInformation) and tagged values from the MDSF UML Analysis Profile.
- Communication Diagram: A class diagram explaining the communication paths and protocols between the system components. The class diagram is annotated with stereotypes (e.g., SystemComponent, Database, Communication) and tagged values from the UML Analysis Profile.

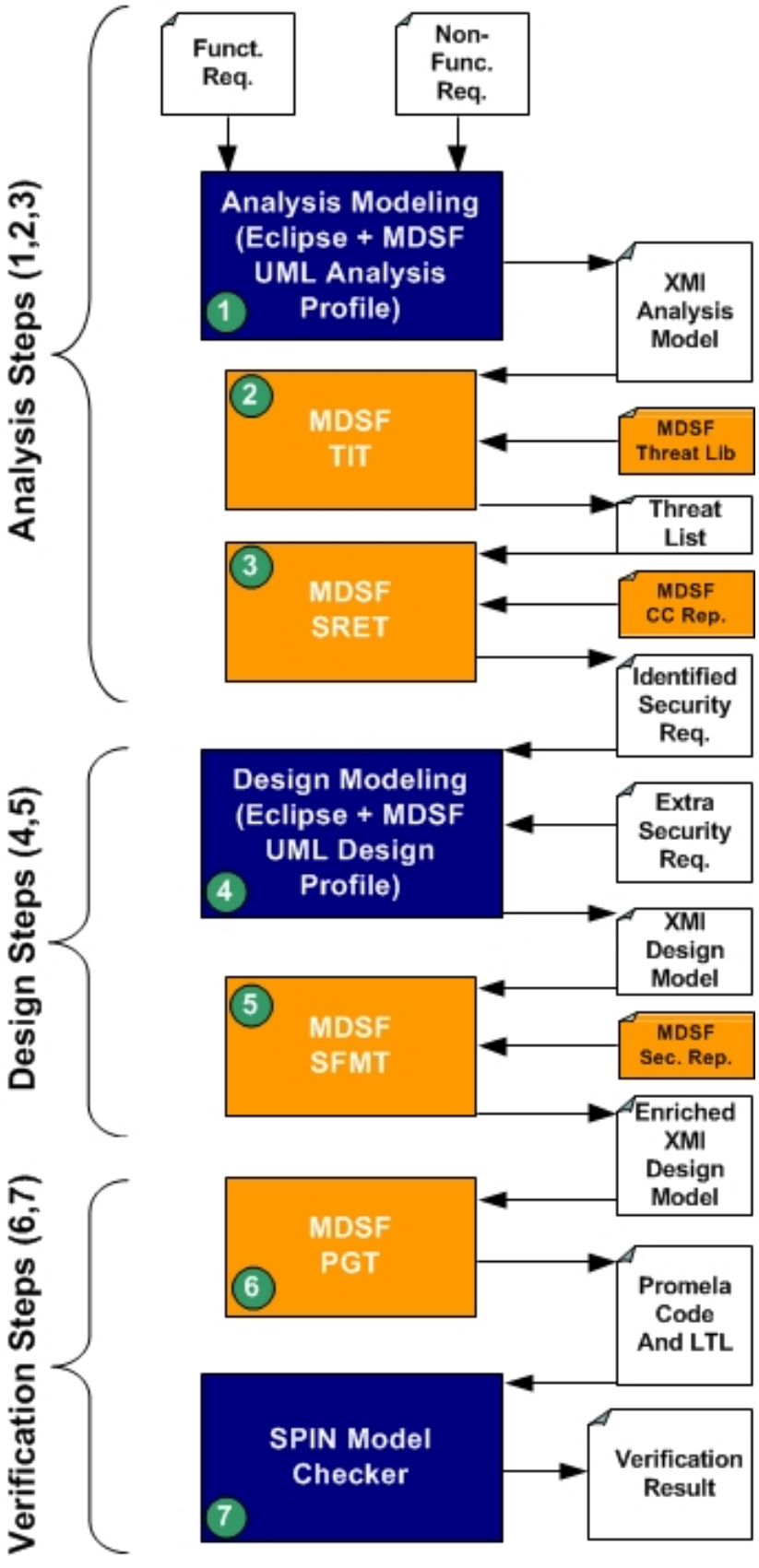


Figure 4.15: MDSF Toolset, Libraries and Their Use

A subset of the MDSF UML Analysis Profile is given in Tables 4.2 and 4.3.

In step 2, the analysis model is input to TIT. TIT is responsible for identifying the possible security threats based on the critical assets, information flow and infrastructure described in the analysis model. Saeki et al. worked on the concept of a threat library in [54] and used the Extended Commercially Oriented Functionality Class for Security Evaluation (E-COFC) [60] as a catalog of threats. E-COFC is an European Computer Manufacturers Association (ECMA) standard, which specifies security evaluation criteria for interconnected IT systems. E-COFC is aligned with the Common Criteria and contains a well-defined list of threats and countermeasures. In MDSF, we use the E-COFC threat catalog as a baseline for our MDSF Threat Library due to its alignment with the Common Criteria. The Threat Library is designed as extendable. Users of MDSF can define their own threats in the MDSF Threat Library to extend the baseline. The structure of the MDSF Threat Library is given as part of Figure 4.35 using a conceptual database schema diagram. This diagram presents Threat and Threat Constraint classes as part of the MDSF Threat Library. Every threat in the MDSF Threat Library has some associated threat constraints. Threat constraints are stored as OCL statements that generate a Boolean result when executed by TIT. TIT executes all OCL statements associated with all threats on the analysis model and identifies existence of a threat based on the Boolean result.

We show an Information Disclosure threat as an example to illustrate how TIT works. Information disclosure may exist if the receiver of a critical piece of information is less trusted than the sender of the same information. TIT executes OCL constraint in Table 4.8 on the analysis model to determine whether information disclosure is a threat for the TOE.

At the end of step 2, the analysts receive a list of the possible threats identified by applying pre-defined OCL invariants on the analysis model. Analysts can also enhance this list with additional threats that they foresee as being possible.

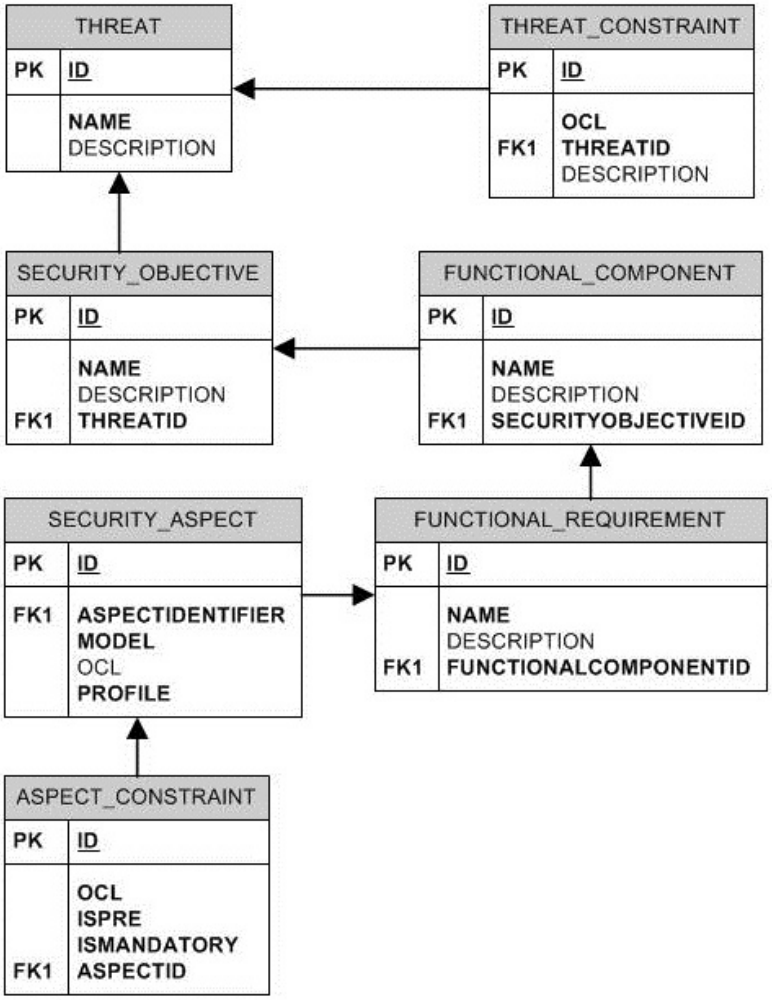


Figure 4.16: MDSF Library: Database Schema for the Common Criteria Repositories

In step 3, the accumulated threat list is input to SRET. As shown in Figure 4.35, the MDSF Repository has a list of SOs for each stored threat, and a list of SFRs for each SO according to the Common Criteria. SRET is responsible for suggesting mitigation activities as SOs. Identified SOs are decomposed into more granular SFRs based on stored information in the MDSF Repository. For the information disclosure threat identified in the previous example, SRET suggests the SOs in Table 4.24 and decomposes each of the SOs into SFRs.

Table 4.24: SOs for Information Disclosure

Security Objective	**Description**
O.Access_Control	The TOE must enforce an access control policy protects information from unauthorized access
O.Authen	Users will be uniquely identified and authenticated prior to performing any actions to be mediated by the TOE
O.Flow_Control	The TOE must enforce an information flow control policy at specified boundaries
O.Resid_Prot	Information stored within the TOE must not be made available to other users upon release
O.Authen_Protect	The TOE must protect authentication information

An example list of SFRs for O.Flow_Control SO is given in Table 4.25 as an example. The Common Criteria uses an operation language that helps construct exact requirements from the SFR templates using operations such as iteration, assignment, selection and refinement. The analyst is responsible for processing these operations and creating final security functional requirements for the TSF.

4.4.1.2. Design Steps. A list of security functional requirements is handed back to the analysts as the result of step 3. These requirements can be enhanced by the help of an analyst and handed to the design team so they can create the design model.

Table 4.25: Example Set of SFRs for O.Flow_Control

Requirement Name	**Description**
FDP_IFC.1.1	The TSF shall enforce the [assignment: information flow control SFP] on [assignment: list of subjects, information, and operations that cause controlled information to flow to and from controlled subjects covered by the SFP]
FDP_IFF.1.1	The TSF shall enforce the [assignment: information flow control SFP] based on the following types of subject and information security attributes: [assignment: list of subjects and information controlled under the indicated SFP, and for each, the security attributes]
FDP_IFF.1.2	The TSF shall permit an information flow between a controlled subject and controlled information via a controlled operation if the following rules hold: [assignment: for each operation, the security attribute-based relationship that must hold between subject and information security attributes]

In step 4, designers create the design model of the software system considering the SFRs. The MDSF UML Design Profile helps the designers annotate and include security properties in the design model. There are two alternatives for enhancing the design model with security functionality. The first alternative is to design the security functionality from scratch by manually using the UML environment and MDSF UML Design Profile. The second alternative is to use MDSF Security Function Repository that contains pre-verified designs for selected security functions corresponding to different SFRs.

An example security function design for FIA_ATD.1.1 SFR of the Common Cri-

teria is given in Figure 4.12. FIA_ATD.1.1 states that a list of security attributes shall be maintained for individual users. These security attributes may include a wide range of parameters such as user identifier, security classification, and clearance level.

Jurjens [18] [19] [20] presented the idea to apply security patterns in UML designs. In MDSF, we enhanced Jurjens' idea with a different design approach, and we propose a mechanism for embedding pre-verified security function designs into the design model of the information system. This approach will help reducing time spent on design and verification steps. The MDSF Security Function Repository stores pre-verified design models for some SFRs. In step 5, MDSF provides SFMT to retrieve pre-verified security function designs from the MDSF Security Function Repository and weave them into the design model. An example model, ready for SFMT, is given in Figure 4.17. The model is annotated with extension points for FIA_ATD.1.1 SFR.

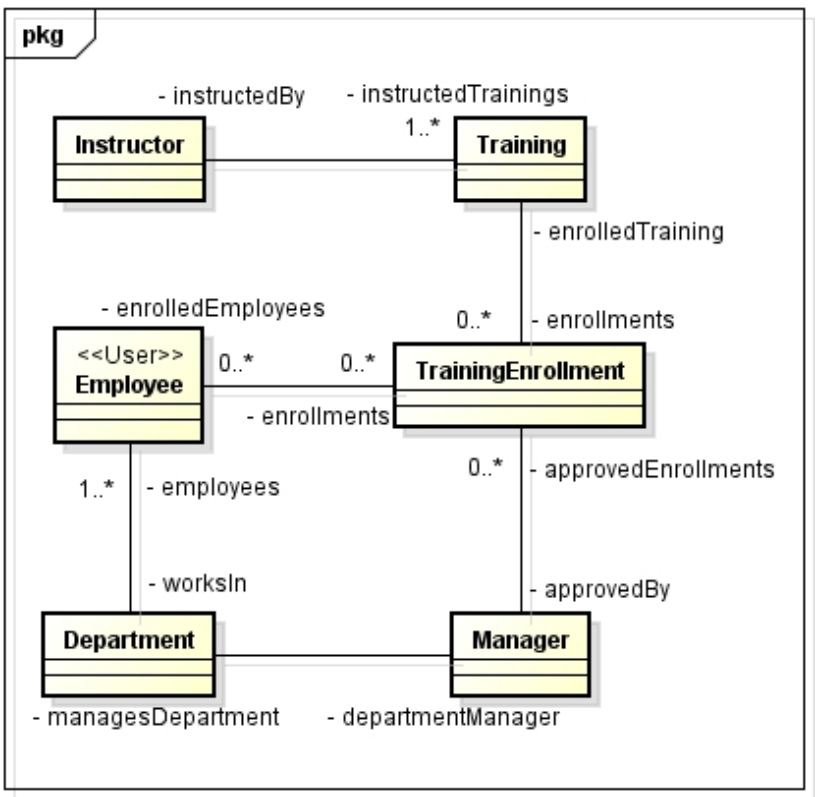


Figure 4.17: Example Design Model as SFM input

SFMT analyzes the input design model and identifies which security functions

from the MDSF Security Function Repository need to be retrieved and woven into the model. SFMT uses the MDSF UML Design Profile to identify the security functions and their integration points. SFMT checks the model for the consistency and annotation of the necessary integration points using the embedded OCL invariants for the security function designs. SFMT identifies the extension point and related security function by examining the stereotypes and tagged values in the annotated design model. After identifying the extension point and related security function, SFMT retrieves the security function from MDSF Security Function Repository, executes the weave operation on the input design model and creates a final design model that includes the security function design. SFMT also checks the final design model using OCL invariants after the merge operation is completed to check whether the model is still consistent. An example SFMT final model generated after FIA_ATD.1.1 merge operation is given in Figure 4.18.

4.4.1.3. Verification Steps. In step 6, the final design model is input to PGT of MDSF. PGT is responsible for transforming UML class diagrams, sequence diagrams and multi-level state diagrams into a specification in PROMELA language. A multi-level state diagram is a type of nested state diagram where each composite state represents a state diagram for one of the classes in the system. An example multi-level state diagram is given in Figure 4.19.

High level operation of the PGT is described in Table 4.26.

PGT creates a separate process for each class with an embedded state machine. These processes communicate with each other with the help of channels and triggers in PROMELA [61]. Each message in the sequence diagram and multi-level state diagram generates a message passing from one process to another or inside the same process. Messages are received by associated channels of the processes and cause a change in the state variable of the class. A state change can also generate a message transfer and cause a state change in one of the processes. PGT also handles the guard conditions and SFR enforcement rules (constraints) written in the OCL language.

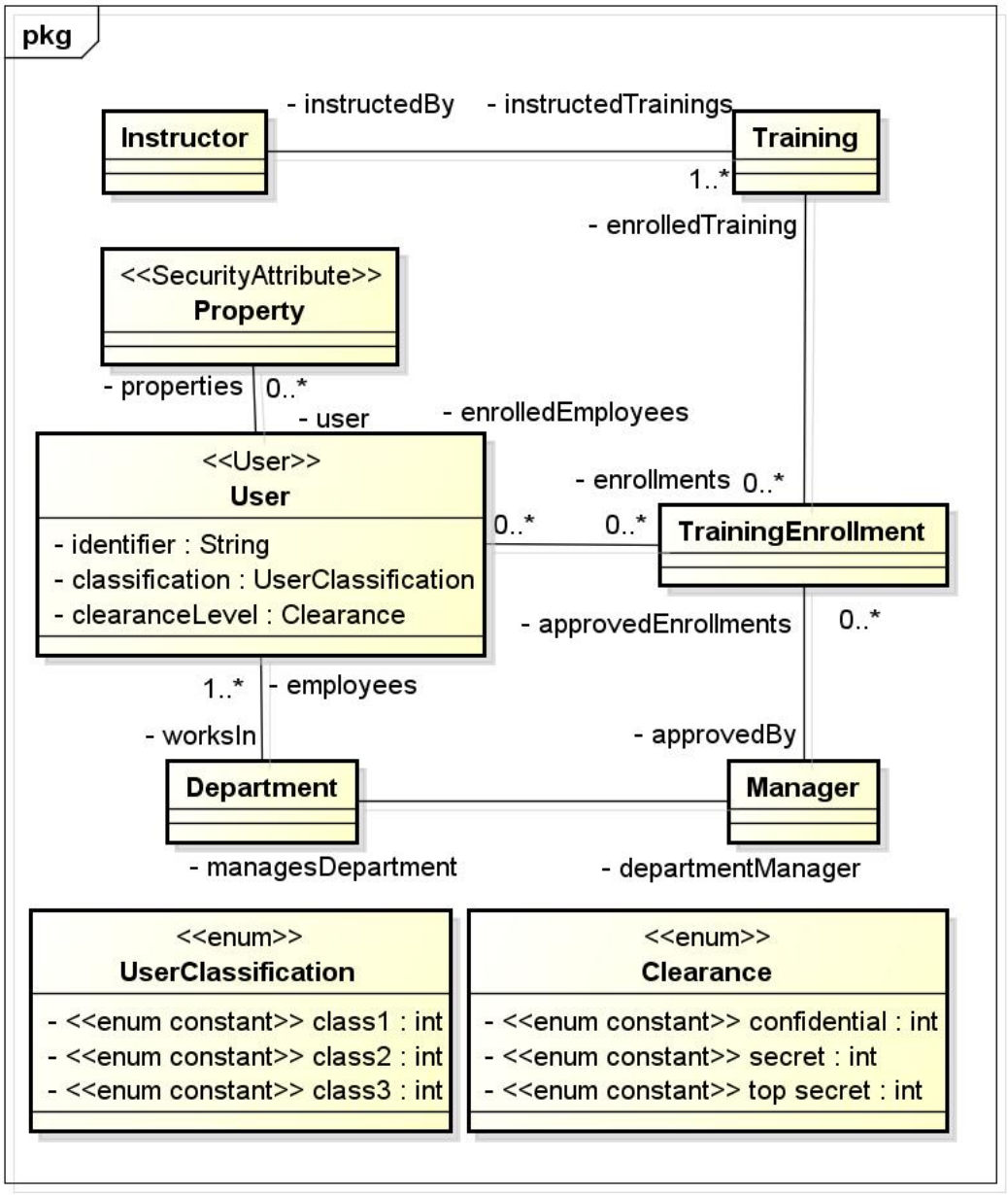


Figure 4.18: Example Design Model as SFM output

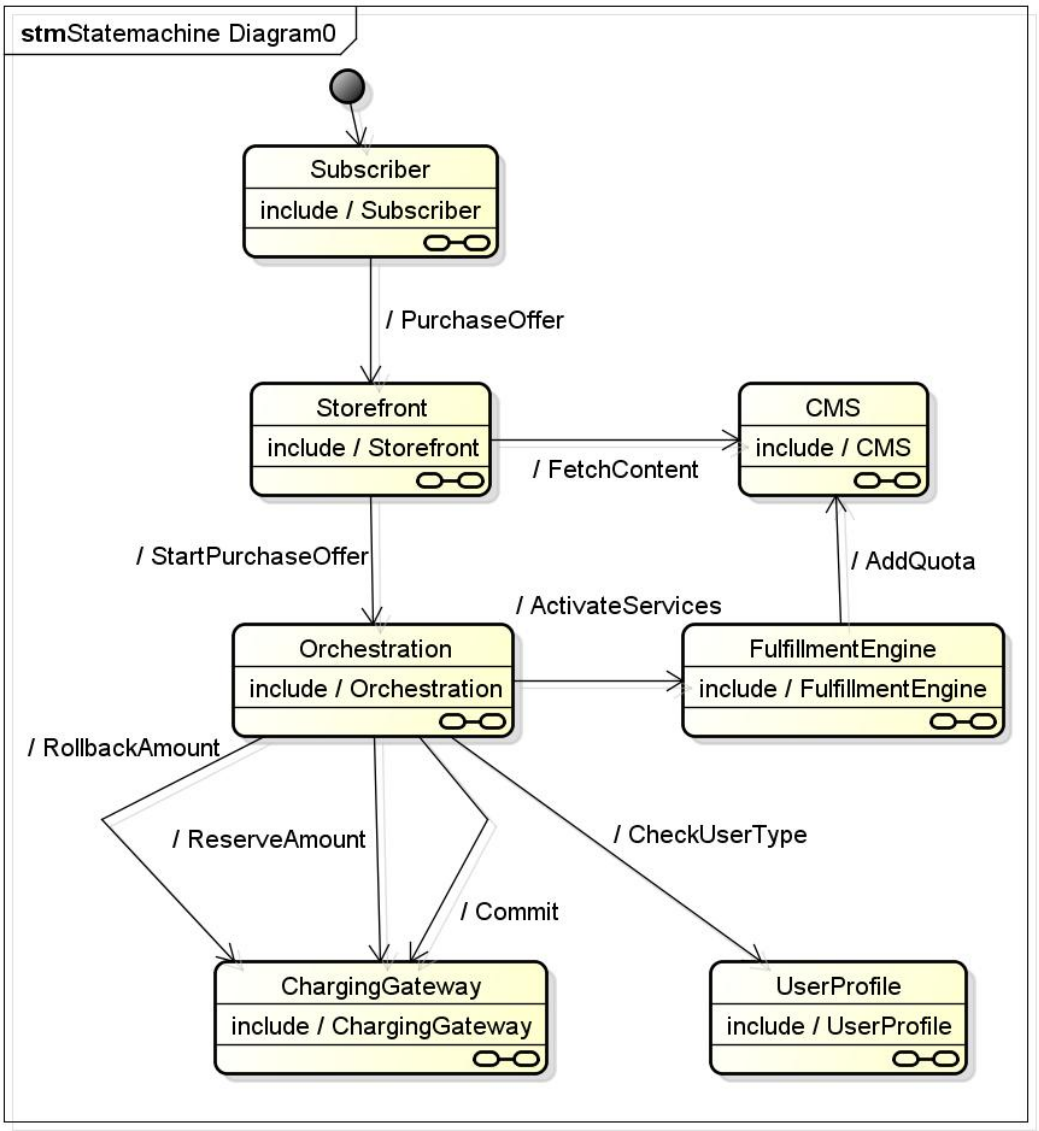


Figure 4.19: Example Multi-level State Diagram

Table 4.26: PROMELA Converter Working Principles

CreatePromelaMTypeForActions(sequence-diagram)
ForAll cls **in** class-diagram
stm **is** GetStateMachine(mlevel-state-machine, cls)
stateList **is** GetStates(stm)
eventList **is** GetEvents(stm)
actionList **is** GetActions(stm)
CreateTriggerChannelForClass(cls)
CreateMTypeForStates(stateList)
CreateGlobalStateForClass(cls)
CreateProcess(stm, stateList, eventList, actionList)
End For
CreatePromelaStateManagerProcess()
RetrieveOpaqueLTL(package)

Guard conditions for state transitions in the state diagrams are converted to conditions in the PROMELA specification. Some of the classes also contain operations that are defined as opaque expressions written in C language. PGT extracts these operations and embeds them inside the PROMELA specification as c_code fragments. The full PROMELA specification of the model can now be formally verified using the Spin Model Checker [44].

In step 7, the PROMELA specification and LTL properties are input to the Spin Model Checker using the XSpin graphical user interface [44]. Spin generates a C program, which performs an exhaustive search in the whole state space of the design model. After the execution of the program, Spin generates counter examples for the properties that are not satisfied during verification. Counter examples can be analyzed

in detail using the simulation capability of Spin Model Checker. Simulation analysis reveals the design errors in the model and helps analysts and designers fix the errors before they are transferred to the source code of the system.

4.4.2. Details of MDSF Tools

4.4.2.1. Threat Identification and Security Requirement Extraction Tools. Security Requirement Extraction Tool is developed to support Threat Auto Detection , and SFR Decomposition processes of the MDSF.

A UML activity diagram is presented to describe how the tool works in Figure 4.20.

After all the initializations are completed, tool checks model integrity constraints on the analysis model. If any of the constraints fail, tool stops execution and provides error message stating that input model is not valid. If the input model satisfies all of the constraints, tool retrieves all threats information from the persistence layer. Every threat has associated OCL constraints. If an OCL constraint evaluates to TRUE, this means that there's a possibility that this threat will exist for the analysis model. With the help of associated OCL constraints, all the threats are identified. Each identified threat is satisfied by one or more security objectives. These security objectives are retrieved from the persistence layer. Every security objective is further decomposed into functional components and security functional requirements with the help of the persistence layer. At the end of the execution, tool provides a list of all possible threats and some security functional requirements, when implemented will help to avoid identified threats.

4.4.2.2. Security Function Merger Tool. Security Function Merger Tool is developed to support Security Aspect Selection and Security Aspect Weaving processes of the MDSF.

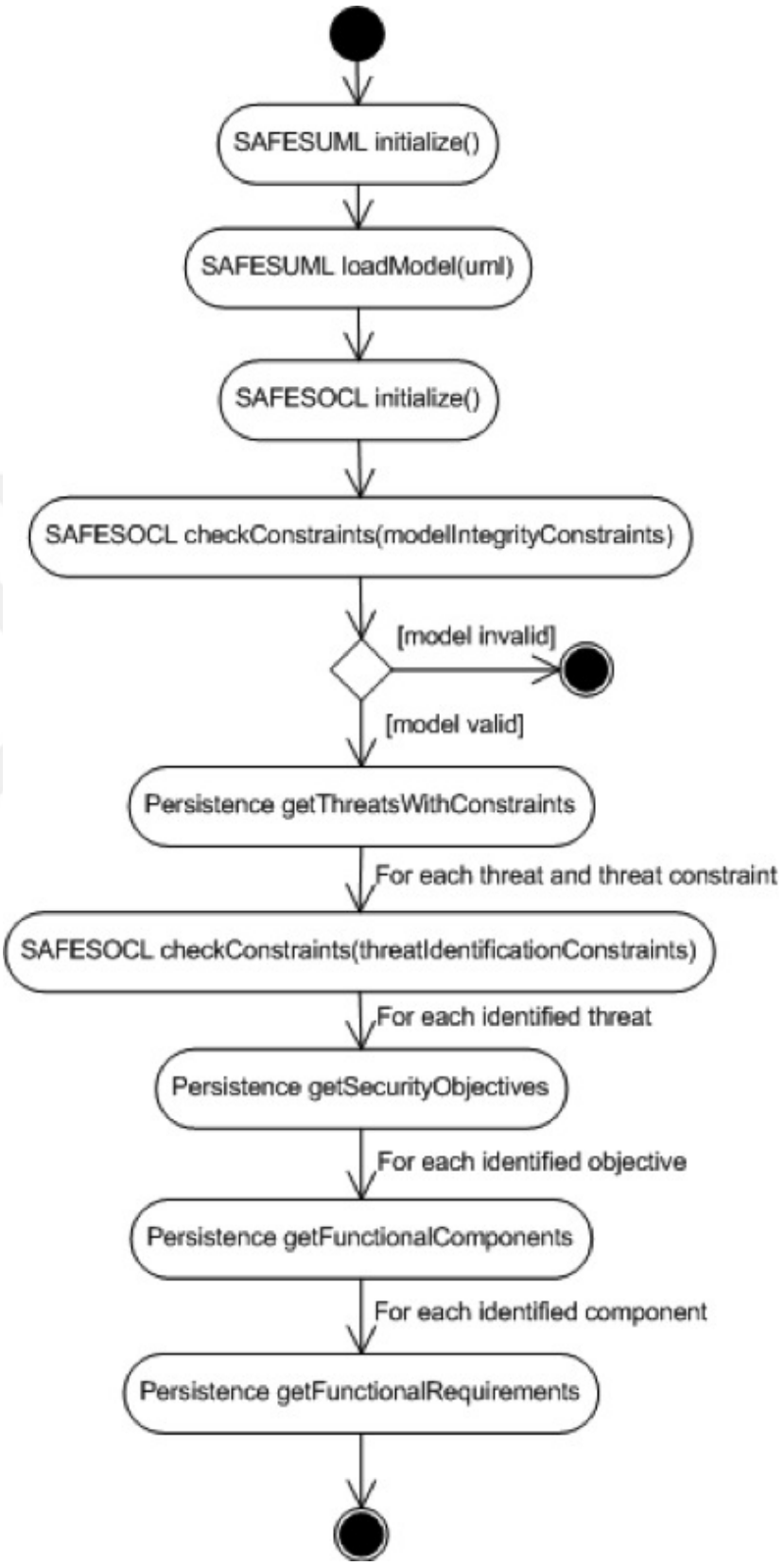


Figure 4.20: UML Activity Diagram for TIT and SRET

A UML activity diagram is presented to describe how the tool works in Figure 4.21.

After the initializations are completed and design model is loaded into the environment, tool gets details of selected security aspect from the persistence layer and loads the security aspect model into the environment. As described in Section 4.3.2, every security aspect has associated pre and post validity constraints. The tool checks mandatory pre-validation constraints on the design model in order to understand if the model is ready to be merged with the security aspect. If any of these constraints fail, tool stops execution and provide an error stating that input design model is not valid. If all mandatory pre constraints are satisfied tool check optional pre constraints on the design model and provide warnings for the optional constraints that are not satisfied. Tool applies security aspect profile to the design model just after the constraint checking operation is completed successfully. After the profile applied, design model is ready to be merged with security aspect model. All enumerations, classes, properties and associations are merged with the design model. During the merge operation, tool identifies the marked extension points (Section 4.3.2) in the design model and creates bindings using generalizations and some other UML mechanisms. After aspect-weaving operation is completed, tool starts checking post validation constraints. These constraints validate if aspect merge operation is completed successfully. If all mandatory post constraints are satisfied, tool saves the design model with all the applied changes.

4.4.2.3. PROMELA Generator Tool. PROMELA Generator Tool is developed to support Model Checking Input Transformation process of the MDSF. This tool is used to convert UML class diagrams and state machines to PROMELA models. After the conversion, these PROMELA models can be used to model check security properties related to dynamic behaviors of software systems. A UML activity diagram is presented to describe how the tool works in Figure 4.22.

After the initializations are complete, PROMELA global variables are extracted

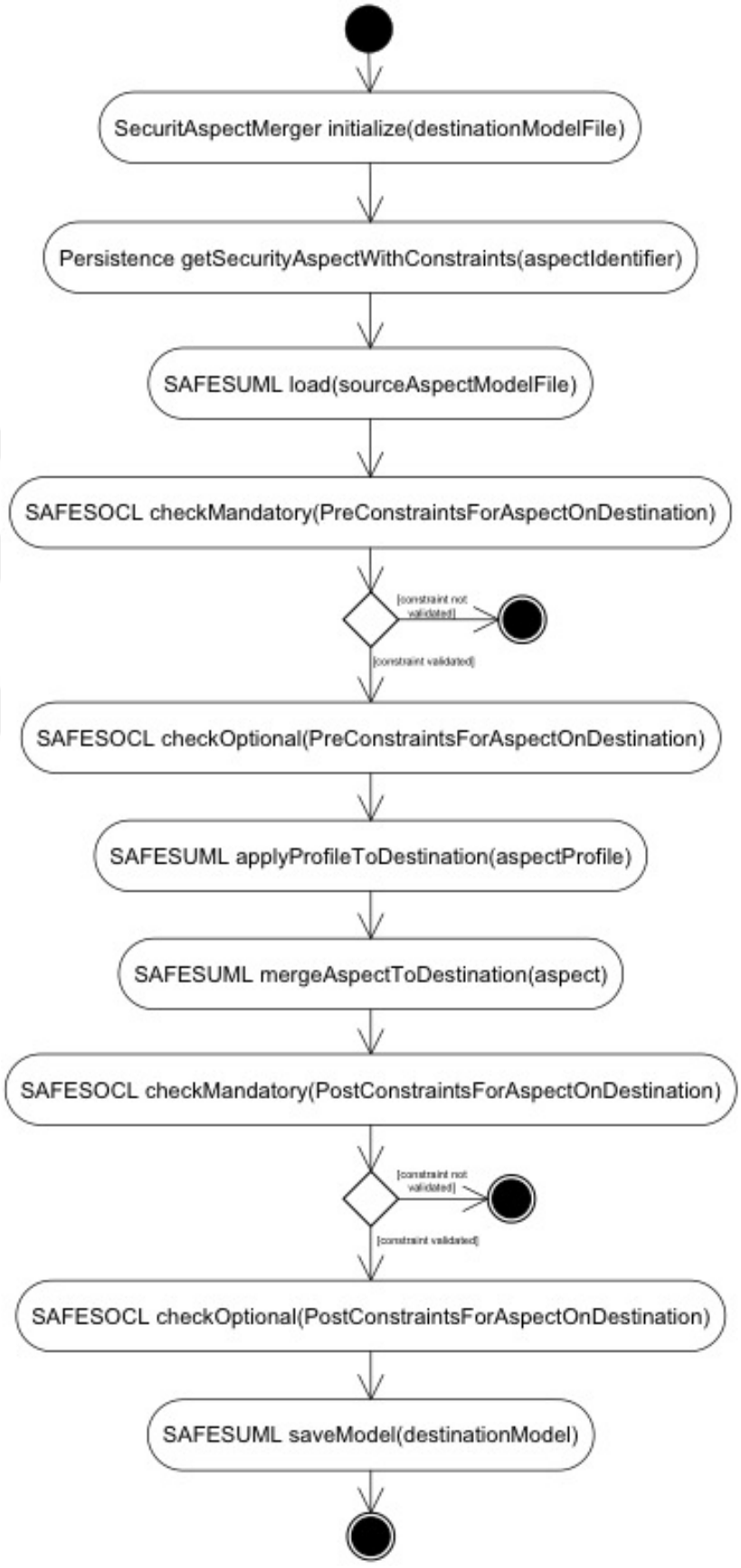
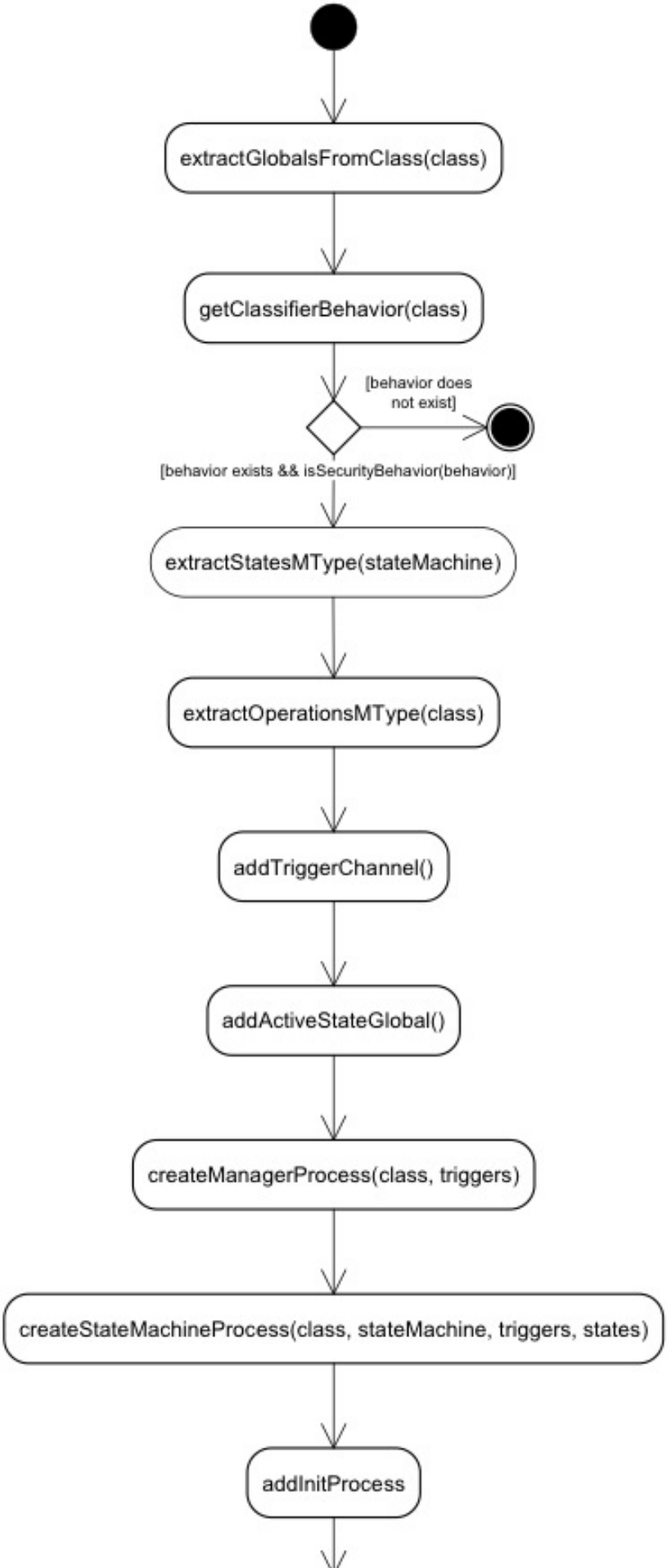


Figure 4.21: UML Activity Diagram for SFMT


extractGlobalsFromClass(class)
getClassifierBehavior(class)
[behavior does not exist]
[behavior exists && isSecurityBehavior(behavior)]
extractStatesMType(stateMachine)
extractOperationsMType(class)
addTriggerChannel()
addActiveStateGlobal()
createManagerProcess(class, triggers)
createStateMachineProcess(class, stateMachine, triggers, states)
addInitProcess

from the class. Application checks if class has defined dynamic behavior and it's marked as a security behavior. If a security behavior for the class exists, PROMELA mtype is defined for all states in the state machine. Another PROMELA mtype is defined for all the operations in the class. A PROMELA channel is defined for the triggers that will be fired and received. A PROMELA variable is defined to track the active state of the object. After all variable and mtype definitions are complete, main process, state machine process and init process are created. These three processes contain all necessary statement to create the state space for the owned security behavior for the class. Init process runs the main process, which is responsible to fire the external triggers towards the state machine. State machine process contains the execution rules extracted from the UML state machine states, transitions, choices, guards and effects. In simplest words, state machine process is created using a model-to-model transformation from UML class and state machine diagrams to PROMELA model.

For some designs, it's not possible to go into enough details using UML diagrams. MDSF provides an additional feature where formal functional definitions can be embedded into UML diagrams using stereotypes and tagged values. These formal definitions can also be used to model check dynamic security properties in more detail. PGT tool is designed to convert UML class diagrams together with their embedded formal function definitions to PROMELA models. A UML activity diagram is presented to describe how the tool works in Figure 4.23.

This tools works on classes, which have operations, annotated with "OperationWithFormalBody". This stereotype shows the tool that there's an embedded formal operation definition within the class diagram. If there's no class with the stereotype, tool exits without any further processing.

Tool checks the operation's "formalExpression" tagged value to get the formal specification body. After all necessary information about class diagram and formal definitions are acquired, tool starts creating the PROMELA model. An active process is created for the formal operation. All attributes of the class are added as global variables to the PROMELA specification in order to use them during model checking.

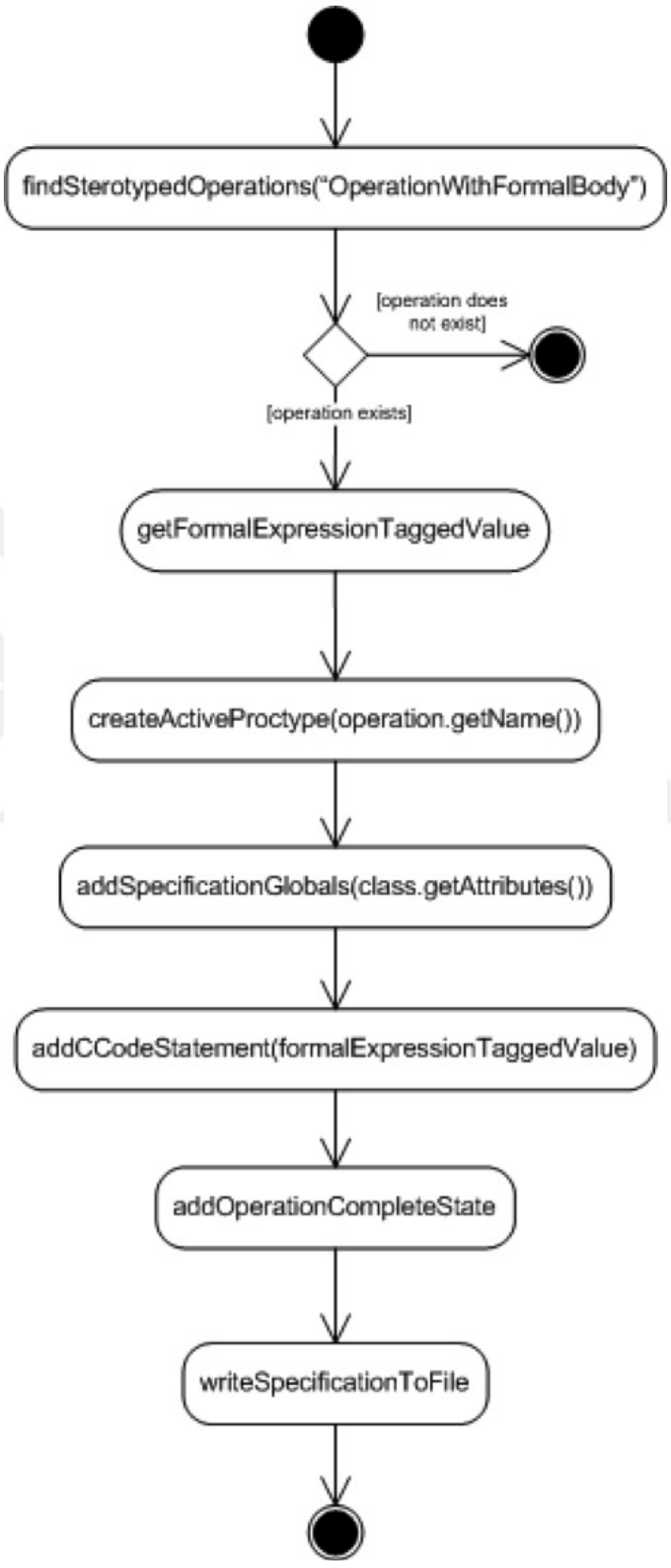


Figure 4.23: UML Activity Diagram for Embedded Formal Definitions

Formal operation body is added to the process specification as a c_code fragment. An operation complete state is also added to end of the process model to be used in LTL specification during model checking. After PROMELA specification is completed, all output is written to a file to be checked in SPIN model checker. A proof of concept case study using this tool is provided in Section 5.

4.4.3. MDSF Toolset Usage Guide

The aim of MDSF processes is to provide a structured way to achieve higher quality security design. For this reason, MDSF provides a process catalog to describe the details and steps of each process. This catalog is also used as a guide during process execution. In order to provide a governance model, and to provide an end-to-end execution framework, a process flow map is also provided.

A short description of every process in the process catalog is provided in below sections. Roles and responsibilities for the processes are provided using a simple RASCI analysis table. Every letter used in these tables defines one of below roles:

- Responsible: These people are the main responsible for the process.
- Accountable: Owner and sponsor of the process
- Supporting: People who are supporting the process with resources
- Consulting: People who can be consulted during process execution
- Inform: People to be informed during the process

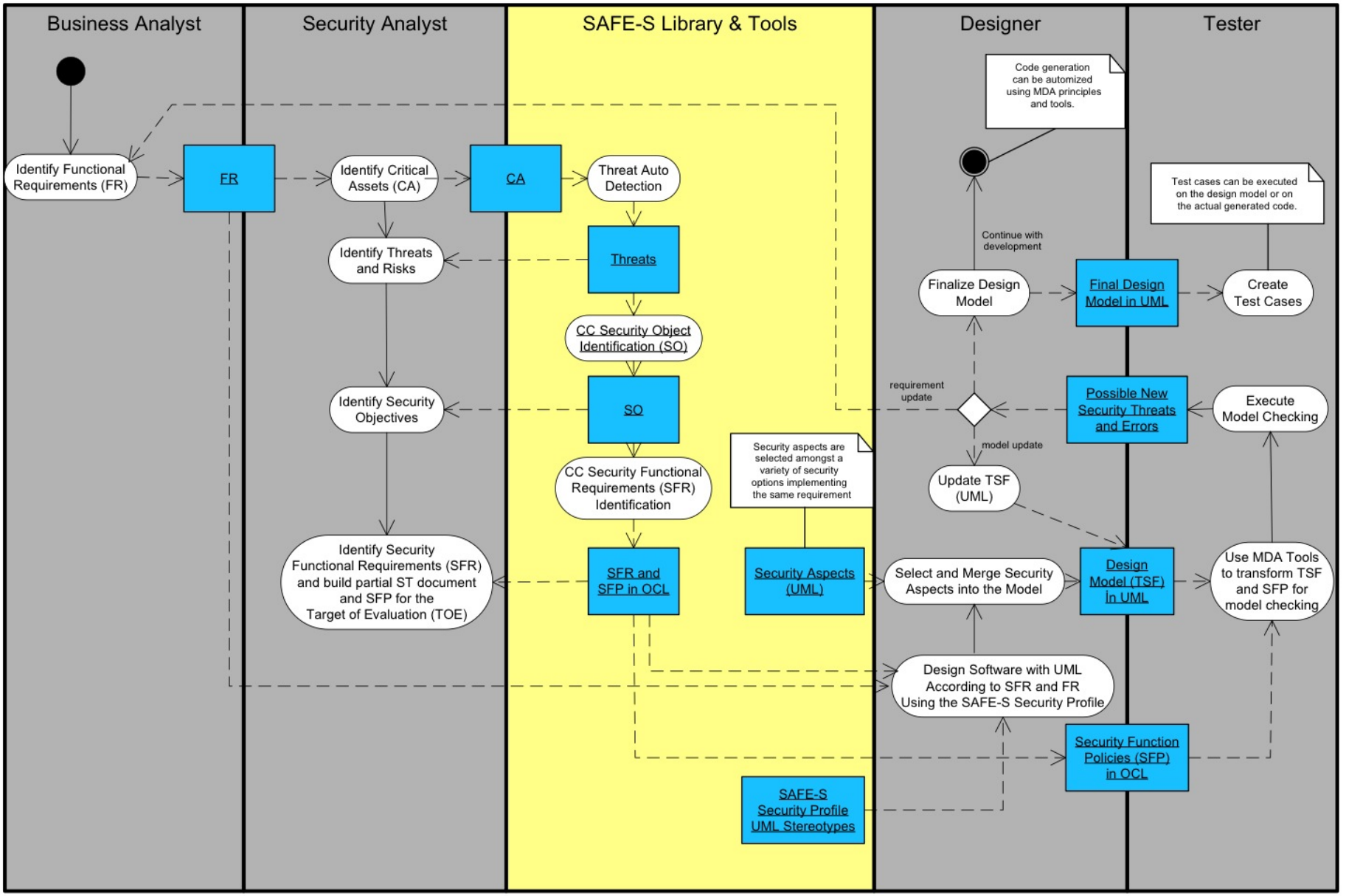


Figure 4.24: Overview of the framework processes

4.4.3.1. Critical Asset Identification. Critical asset identification is the first process in the MDSF process catalog. The aim of this process is to identify assets, which may cause vital, financial or any other problems when the system is under attack by hostile forces. Input to this process is only a high level UML class diagram that shows system components, information stored, processed and transferred between the system components and their relations.

Table 4.27: Critical Asset Identification RASCI Analysis

	Analyst	Sec. Expert	Designer	Req. Owners	Developers	Tester
Critical Asset Identification	**R**	**S,C**		**A,I**		

Analysts are responsible to identify the critical assets on the analysis model's class diagram. MDSF provides necessary UML profile for critical identification process. Details of the profile can be found in Section. Mainly, "Asset" stereotype and "DataClassification" tagged value will be used for this purpose.

For an example of critical asset identification process and models please refer to Section 5.

4.4.3.2. Threat Auto Detection. This process helps the team to identify possible threats using the analysis UML models. The list of extracted threats should be used as a guide where security experts using none-automated processes can identify several other threats. This process requires the analysis UML class diagram and asset identification UML diagram created by the critical asset identification process. In addition to these UML diagrams, description of critical system use cases defined in UML sequence diagrams is required for the threat auto detection. List of required UML diagrams are described in Table 4.28.

Communication diagram is a new diagram created only to help threat auto de-

Table 4.28: Threat Auto Detection Required Diagrams

Diagram Name	**Description**	**Diagram Type**	**Created In**
Class Diagram	Diagram generated as part of the analysis UML model	UML Class Diagram	Functional Requirements Analysis
Critical Asset Diagram	Diagram annotated for identification of critical assets	UML Class Diagram (with MDSF Profile)	Critical Asset Identification Process
Use Case Descriptions	Diagram describing the flow of information between system components for specific use cases	UML Sequence Diagram (with conveyed information)	Functional Requirements Analysis
Communication Diagram	Diagram annotated for identification of communication paths between system components	UML Class Diagram (with MDSF Profile)	Threat Auto Detection

tection process. This diagram uses MDSF UML profile for identification of types of system components and communication paths between them.

Table 4.29: Threat Auto Detection RASCI Analysis

	Analyst	**Sec. Expert**	**Designer**	**Req. Owners**	**Developers**	**Tester**
Critical Asset Identification	**S,C**	**R**		**A,I**		

These three diagrams together with the applied profile, contains all necessary information to analyze and identify some possible threats from the available threat library. As described in Table 4.29, security expert is responsible to input these four

diagrams (Table 4.28) to the threat identification tool in XMI format. Tool automatically selects appropriate threats from MDSF Threat Library (Section 4.3.3) with the help of built in rules written in OCL (Section 4.3.1.1). Tool is flexible such that new threats and rules can be inserted manually.

Framework only suggests possible threats; however security expert is still responsible to filter and prioritize the suggested threats and include additional threats, which cannot be identified with the automated process.

4.4.3.3. Security Objective Decomposition. Aim of this process is to identify security objectives, which will help software designers and security experts to mitigate possible threat identified in Threat Auto Detection process.

For the identification of security objectives, MDSF provides security objective and SFR decomposition library (Section 4.3.4). This library contains a mapping of threats to security objectives. After the possible threats are identified by the threat auto detection process and enhanced by the security expert, these threats are converted to related security objectives with the help of the library.

Table 4.30: Security Objective Decomposition RASCI Analysis

	Analyst	**Sec. Expert**	**Designer**	**Req. Owners**	**Developers**	**Tester**
Critical Asset Identification	**S,C**	**R**	**I**	**A,I**		

Security expert is also supposed to include other security objectives where it's appropriate and not identified by the MDSF library and tools. At this stage, it's also important to keep the designers in the information loop since security objectives will also be an important part of the design process.

4.4.3.4. SFR Decomposition. The aim of this process is to further decompose identified security objectives to security functional requirements. MDSF provides Security Objective and SFR Decomposition Library (Section 4.3.4) which contains necessary functional components which satisfy the identified security objectives. Functional components are security functional requirement groupings. As an example, logon limitation security objective (O.Logon_Limit) is satisfied by basic authentication failure handling functional component (FIA_AFL.1).

In the Common Criteria Part 2, FIA_AFL is defined as below:

"FIA_AFL.1 Authentication failure handling requires that the TSF be able to terminate the session establishment process after a specified number of unsuccessful user authentication attempts. It also requires that, after termination of the session establishment process, the TSF be able to disable the user account or the point of entry (e.g. workstation) from which the attempts were made until an administrator-defined condition occurs."

In this process, all security objectives are decomposed into the functional components as described in the above example. Then, these components are automatically expanded into security functional requirements using the Common Criteria repositories included in the MDSF Library.

Continuing from above example, FIA_AFL.1 component is decomposed into two security functional requirements:

- FIA_AFL.1.1 The TSF shall detect when [selection: [assignment: positive integer number], an administrator configurable positive integer within [assignment: range of acceptable values]] unsuccessful authentication attempts occur related to [assignment: list of authentication events].
- FIA_AFL.1.2 When the defined number of unsuccessful authentication attempts has been [selection: met, surpassed], the TSF shall [assignment: list of actions].

During the execution of the process all security functional requirements associated with the identified security objectives are presented to the security expert with the help of the tools. All together, these security functional requirements will define a system, which is resilient, and not vulnerable to the possible threats identified in threat auto detection process (Section 4.4.3.2).

Table 4.31: SFR Decomposition RASCI Analysis

	Analyst	Sec. Expert	Designer	Req. Owners	Developers	Tester
Critical Asset Identification	**S,C**	**R**	**I**	**A,I**		

The Common Criteria use an operation language, which is used to construct exact requirements from the security functional requirements definitions. As shown in the above example, functional requirements contain some operations like iteration, assignment, selection and refinement. Security analyst is responsible to process these operations and create final security functional requirements for the TOE. For the above example, these final security requirements can be identified:

- The TSF shall detect when 3 unsuccessful authentication attempts occur related to login use case.
- When the defined number of unsuccessful authentication attempt has been surpassed, the TSF shall redirect to an error page.

4.4.3.5. Security Policy Translation. Security functional requirements are converted to the formal security policies to be used in the verification process. Functional design of the TOE will be formally verified according to the generated formal security policies. This process will help to ensure that TOE satisfies the security requirements and hence is not vulnerable to the identified threats.

4.4.3.6. Design with Security. Designer creates the TSF design model using the MDSF UML Profile and according to functional requirements (FR) and SFR. It's also possible to enhance the TSF design model with the help of security aspects retrieved from MDSF Library. These aspects are pre-verified design models, which satisfy a specific set of SFRs. TSF design model can also be modified after the results from formal verification is received.

4.4.3.7. Security Aspect Selection. After the security functional requirements are identified in SFR Decomposition process, it's possible to select some security aspects directly from the MDSF security aspect library.

Please refer to Section 4.3.2 for a complete list of provided security aspects. Aspect selection can be done automatically by Security Function Merger Tool (Section 4.4.2.2) or it can be handled manually by the security designer.

4.4.3.8. Security Aspect Weaving. If security designer decides to use some pre designed security aspects from the MDSF security aspect library, and selects appropriate security aspects, it's Security Function Merger Tool's responsibility to merge this aspect to the main design model of the software in consideration. There are two main steps for security aspect weaving operation:

- Stereotyping the design model
- Execution of Security Aspect Merger tool

In order to identify the binding points between the main design model and security aspect design model it's necessary to annotate the main design model with necessary stereotypes. Please refer to Section through Section for further details on security aspect profiles and binding points.

After the binding points are identified on the design model, it's input to the security aspect merger tool together with the identifier of the selected security aspect.

Tool automatically merges two diagrams and binds them together with the help of the stereotype annotations. For further details on the Security Aspect Merger Tool, please refer to Section 4.4.2.2.

4.4.3.9. Model Checking Input Transformation. As described earlier in the document there are two types of verification techniques used in MDSF:

- Static verification using OCL execution on static UML diagrams
- Dynamic verification of behavioral security properties on state machines

The second option, dynamic verification, requires formal verification techniques to be applied in order to have a complete verification. It's also possible to utilize UML execution engines and apply simulation based checking; however it's not possible to cover all state space of the application with simulation.

MDSF toolset provides an application, which converts UML class diagrams and behaviors specified within state machine diagrams to PROMELA models. These PROMELA models can easily be model checked using SPIN model checker. In addition to bare model checking, LTL properties can be checked on the PROMELA model.

MDSF contains a security aspect library corresponding to the security functional requirements extracted from the Common Criteria. All security aspects are modeled using UML class diagrams and state machine diagrams. In order to check if designed security aspect satisfies the corresponding security functional requirements all necessary OCL invariants are defined within the framework. In addition to verification of static requirements, MDSF defines LTL properties for the security aspects to be verified using model checking on the design model of the security aspect. All security aspects are verified using these techniques.

The same methodology and toolset used for security aspects in the library can also be used to model check security designs, which does not use security aspect library. It's expected that proper stereotypes from UML profile (Section 4.2.2) are applied to the diagrams. It's possible to execute OCL statements defined for security functional requirements using the MDSF toolset. In addition to static verification, dynamic security behavior can also be verified using PROMELA Generator Tool (Section 4.4.2.3) and SPIN model checker. For PROMELA conversion to work properly, class in consideration has to have a classifier behavior, which is stereotyped with "SecurityBehavior" stereotype. Please refer to Section 4.2.2 for further details on the defined UML profile.

PROMELA Generator Tool converts only class diagrams and state machine diagrams to a PROMELA model. A class is converted to a PROMELA process. All properties stereotyped with "SecurityAttribute" stereotype are converted to variables in PROMELA model. It's expected that security behavior of the class be defined within a UML state machine diagram. This state machine diagram is converted to several PROMELA statements, which will create the same state space during model checking. Below transformations are handled during PROMELA model creation:

- All transition triggers in the UML state machine model are converted to mtypes in PROMELA model. Only the Call Event triggers are accepted for conversion. These triggers are fired from a manager process within the PROMELA model to mimic the execution of the state machine.
- All states, initial state and final state are converted to mtypes in PROMELA model. Pseudo states are not converted since they only represent middle states such as choice states.
- All effects for transitions are converted to PROMELA statements, which modify variables. Only the properties stereotyped with "SecurityAttribute" are allowed to be part of effect statements.
- All guards of transitions are converted to PROMELA expressions. Only a small subset of OCL language is accepted for conversion to PROMELA expressions.
- All incoming and outgoing transitions for choice pseudo states are converted

to nested PROMELA conditions using the OCL guard statements on top of those transitions. Inner bodies of the nested condition statements are created as PROMELA statements, which modify the current state global variable.

- All transitions which have source and target as real states (not a psedo state) are converted to PROMELA statements, which modify the current state global.
- Transition to final state is converted to a PROMELA break statement, which ends the execution of state machine and main processes.

In order to describe how model transformation works, a sample design is described in Figure 4.25 and Figure 4.26.

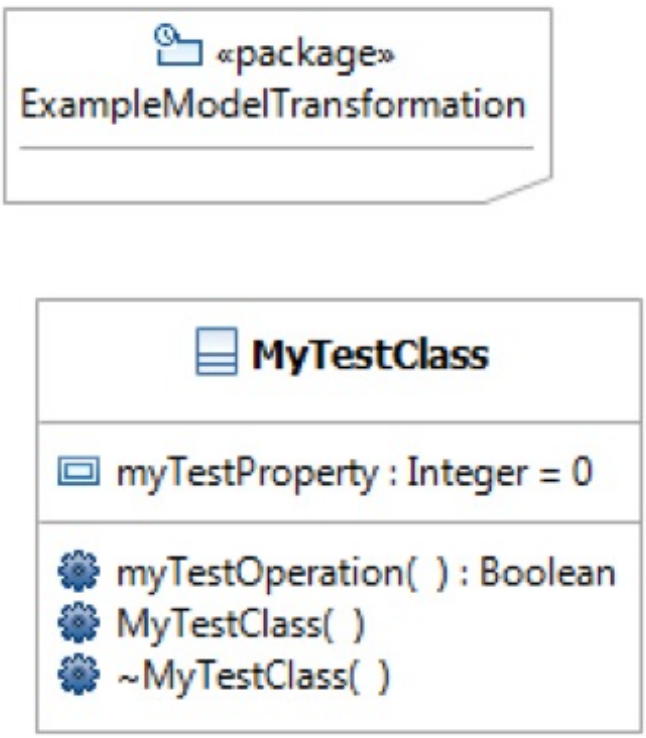


Figure 4.25: Sample class diagram

Above class diagram contains one class ("MyTestClass") which has only one property. "myTestProperty" is an integer parameter which has default value of 0. "MyTestClass" contains a constructor method, a destructor method and a sample operation ("myTestOperation") which returns a Boolean result.

Figure 4.26 describes the dynamic behavior presented by “MyTestClass”.

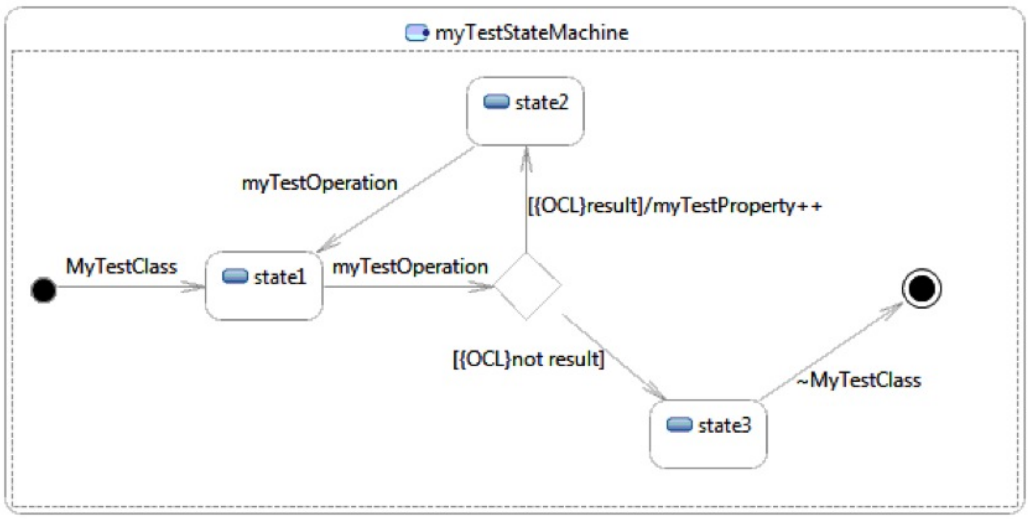


Figure 4.26: Sample state machine

Class enters into “state1” after constructor method is called. At state1, if “myTest-Operation” is called, a choice state is reached. At the choice state, if result of the called operation is true, “myTestProperty” is incremented by one and state2 is activated. At the choice state, if result of the called operation is false, state 3 is activated. At state2, if “myTestOperation” is called, state1 is activated again. At state3, if destructor method is called, final state is reached.

Below figure demonstrates the contents of XMI package, which represents the UML class and state machine diagrams.

Whenever above XMI file is input to the PROMELA Generator Tool, PROMELA specification in Table B.1 is created based on the model transformation rules described in this section.

4.4.3.10. Model Checking. Model checking is a complementary process in secure software development lifecycle. If any requirement is not satisfied in the design phase or somehow inappropriate security aspects are merged into the design, model checking

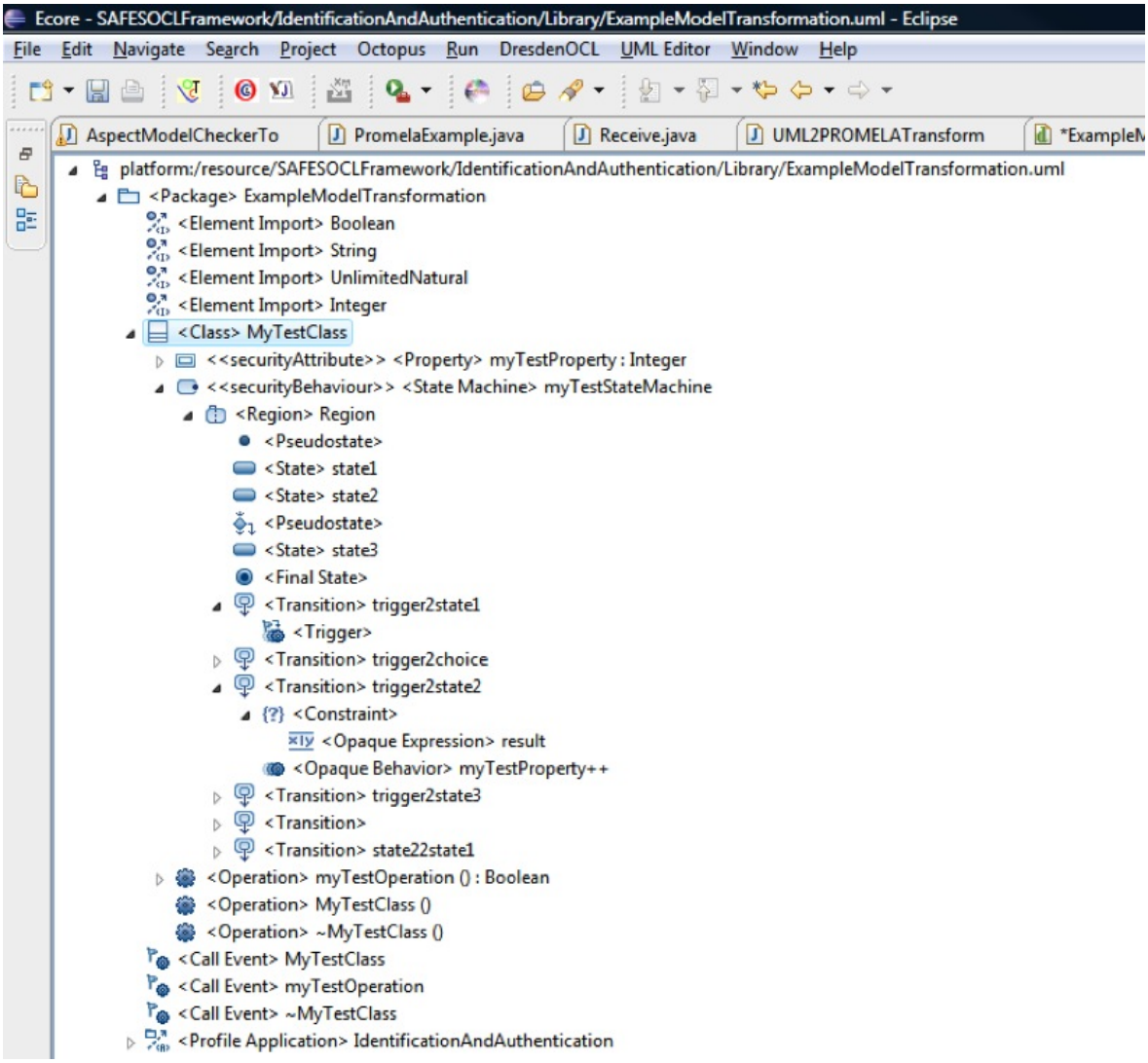


Figure 4.27: XMI structure of class diagram and state machine diagram

can reveal misuse or security holes in the design.

SPIN is selected as the model-checking tool because it's generic enough to model-check a wide range of software systems and has extensive community support.

After the design model is converted to PROMELA with the help of the PROMELA Generator Tool, it's possible to check the state space of the application for design flaws. Continuing from the previous sample, PROMELA model in Table B.1 will be checked for problems. Figure 4.28 shows a screenshot of XSPIN and the sample PROMELA model loaded.

There are two options in SPIN model checker; simulation and verification.

Simulation just executes the model and creates a single trail. Within MDSF, simulation is used just to check if model is generated correctly and executes without problems. Another usage of simulation is to visualize the trails (Figure 4.29). After verification, if any security property is not satisfied, a trail is generated by SPIN. This trail represents the execution steps, which cause the error condition to happen. This output trail can be visualized and traced using the simulation feature of SPIN model checker.

Without any LTL properties, SPIN model checker checks safety and liveness properties. Within MDSF, SPIN verification is used to check some LTL properties on the PROMELA model. Continuing from the sample PROMELA model, we may want to check a LTL property as defined below:

[] ((p) -> (q))
#define p (state==STATE3)
#define q (myTestProperty!=0)

With this property, designer wants to check that whenever MyTestClass is in state3, object's myTestProperty property value will be different then 0. This LTL

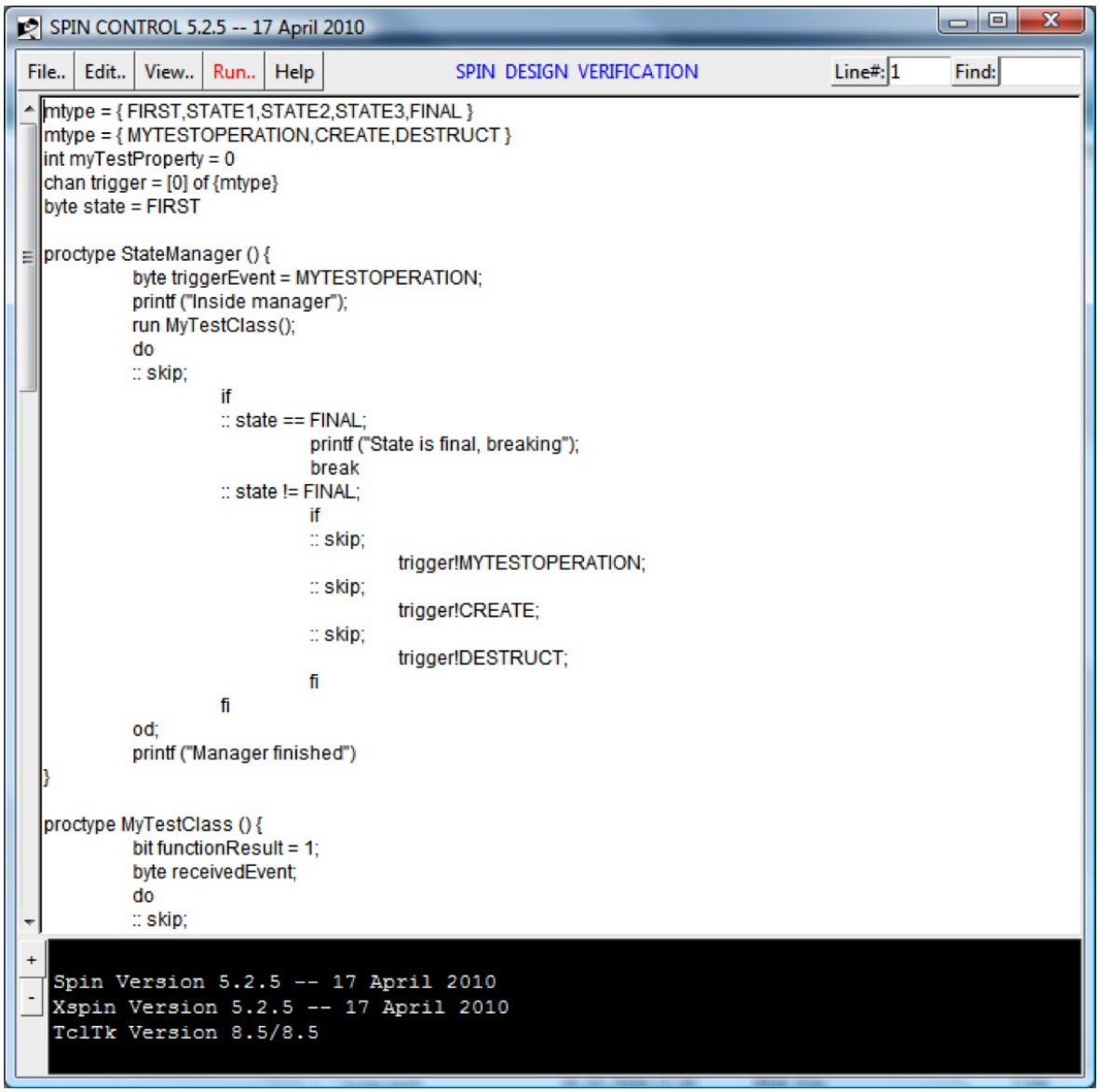


Figure 4.28: PROMELA model in SPIN model checker (XSPIN)

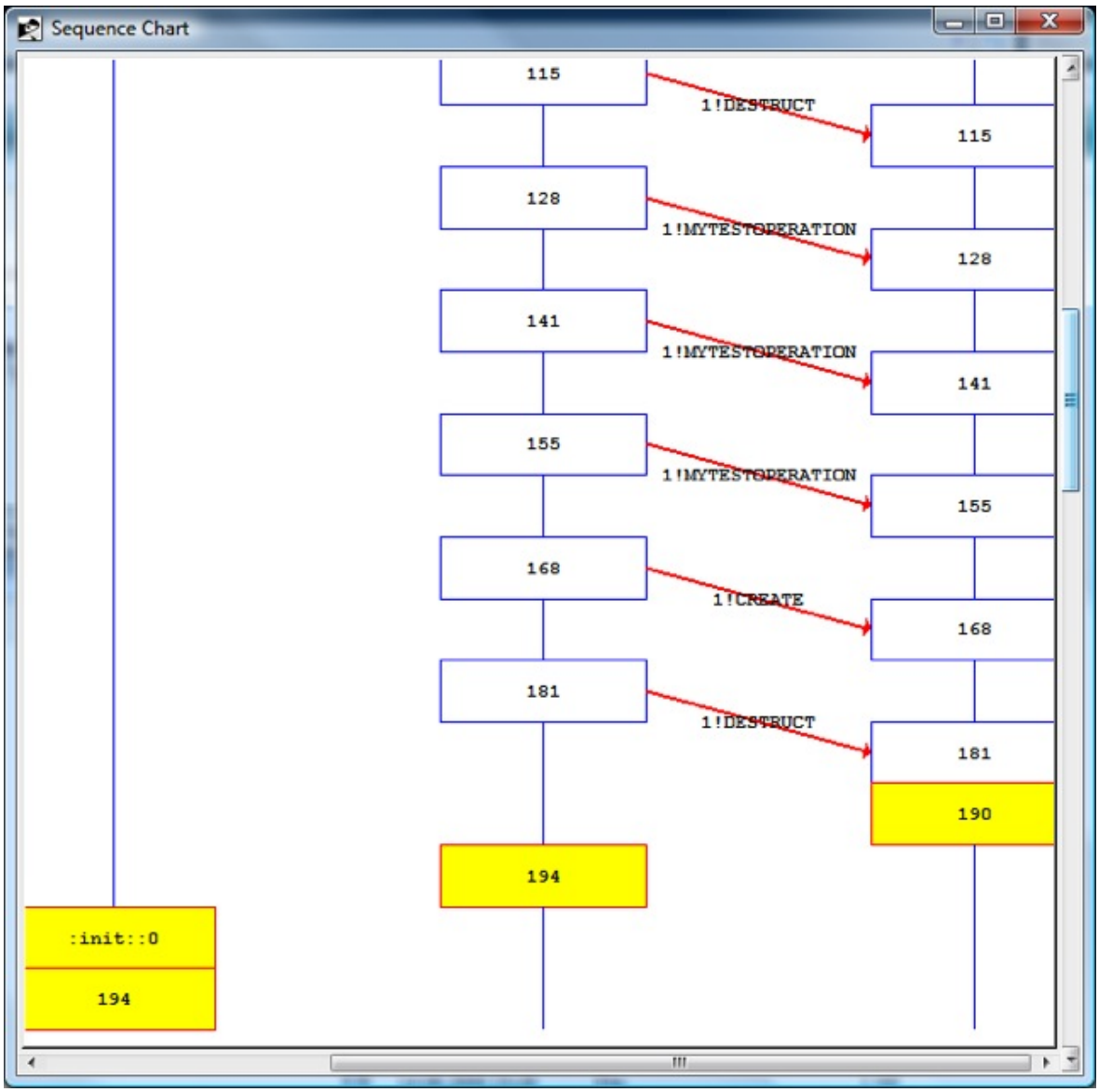


Figure 4.29: Sample XSPIN Simulation Trail

property is converted to a never claim by the tool as defined below:

```
never { /* !([] ((p) -> (q))) */
T0_init:
        if
        :: (! ((q)) && (p)) -> goto accept_all
        :: (1) -> goto T0_init
        fi;
accept_all:
        skip
}
```

When verification is executed in SPIN model checker, a text result is generated which shows that an error condition exists in the model together with the applied never claim.

```
State-vector 32 byte, depth reached 56, errors: 1
        21 states, stored
        2 states, matched
        23 transitions (= stored+matched)
        23 atomic steps
hash conflicts: 0 (resolved)
```

Another output of SPIN model checker is the trail file. Trail file contains the steps of the execution path, which ends in a state where LTL property fail. SPIN has a tool to visualize the output trail as shown in Figure 4.30.

Above trail shows that a new “MyTestClass” is created and directly after creation “myTestOperation” operation is called and operation resulted in “false” result. Creation event puts the object in state1 as shown in Figure 4.26. Calling of “myTestOperation” puts the object into choice pseudo state. “False” result of “myTestOperation” triggers a transition to state3 directly because of the guard condition ([OCL not re-

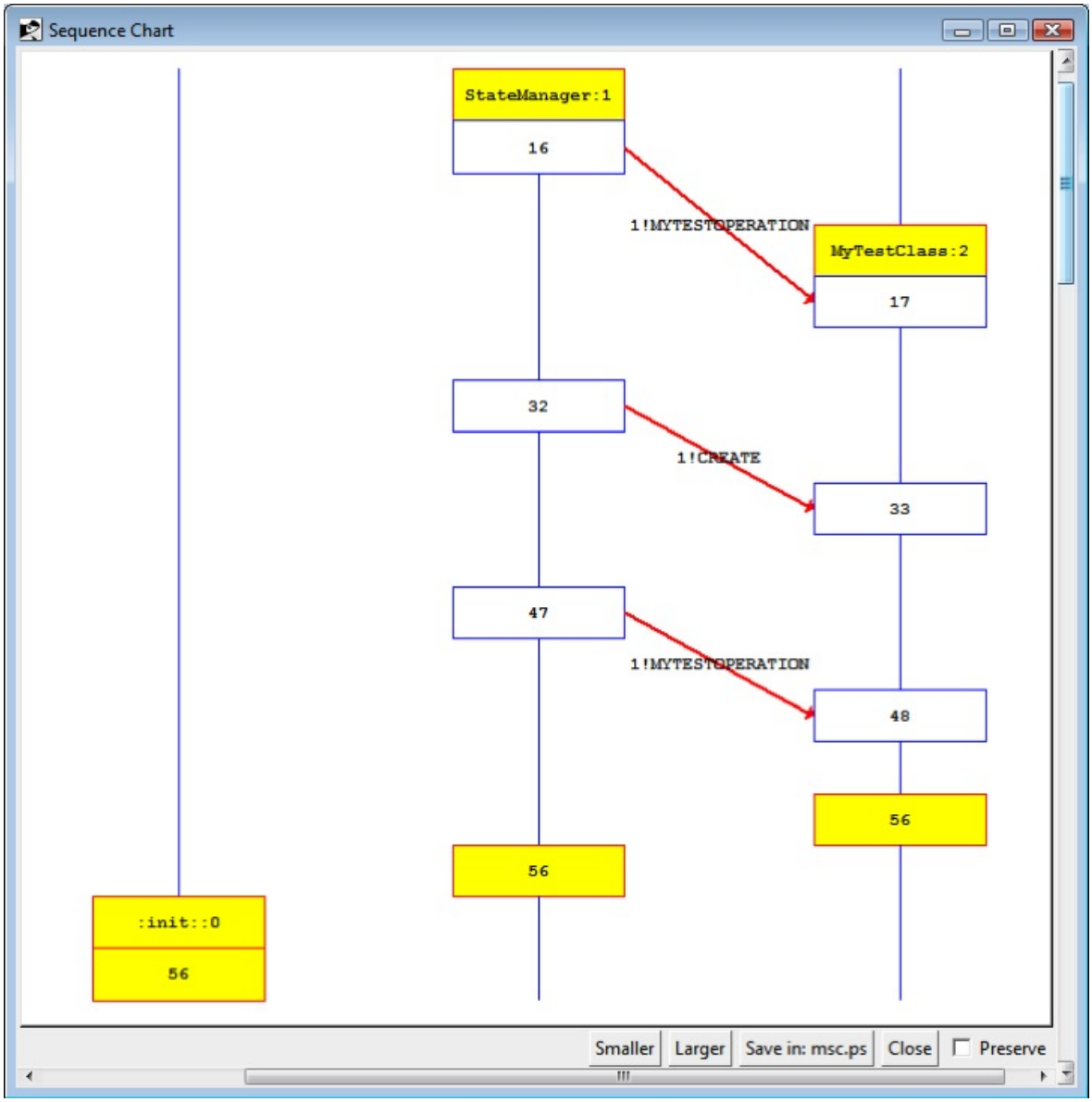


Figure 4.30: Sample SPIN Trail Reflecting Error Condition

sult]). During this execution path, value of “myTestProperty” has never been changed and it still has its default value of 0. This situation invalidates the LTL property which states that being in state3 implies that “myTestProperty” to be other than 0. Execution log (Table 4.32) also helps in double checking the meaning of the output error trail.

Table 4.32: Execution Log of Sample Verification

Starting StateManager with pid 2
Inside manager 4: proc 1 (StateManager) line 9 ”pan_in” (state 1) [printf(’Inside manager’)]
Starting MyTestClass with pid 3
CREATE event received 35: proc 2 (MyTestClass) line 59 ”pan_in” (state 28) [printf(’CREATE event received’)]
State is STATE1 37: proc 2 (MyTestClass) line 78 ”pan_in” (state 49) [printf(’State is STATE1’)] <
MYTESTOPERATION event received 50: proc 2 (MyTestClass) line 44 ”pan_in” (state 10) [printf(’MYTESTOPERATION event received’)]
49: proc 2 (MyTestClass) line 40 ”pan_in” (state 4) [functionResult = 0]
State is STATE3 53: proc 2 (MyTestClass) line 82 ”pan_in” (state 53) [printf(’State is STATE3’)] <

A proof of concept case study showing how above approach is applied to security properties is presented in Section 5.

4.4.4. SDK for Extending MDSF

A modeling application framework has been developed on top of Eclipse Model Development Tools (MDT). UML2 and OCL implementations are used for this purpose. UML2 is an EMF-based (Eclipse Modeling Framework) implementation of the UML 2.0 OMG meta-model for the Eclipse platform. OCL is the implementation of the Object Constraint Language (OCL) implementation of the OMG standard for EMF-based models.

Application framework contains two main libraries:

- SAFESUML: Provides necessary functions to easily load, parse, surf and modify UML models specified in XMI (XML Meta data Interchange) format.
- SAFESOCL: Provides necessary functions to load and validate constraint files written in OCL language.

4.4.4.1. SAFESUML Library Class. SAFESUML class (Figure 4.31) is a wrapper on top of Eclipse EMF UML2 implementation. It simplifies the initialization of the UML environment and provides simple methods to load a UML model into the environment.

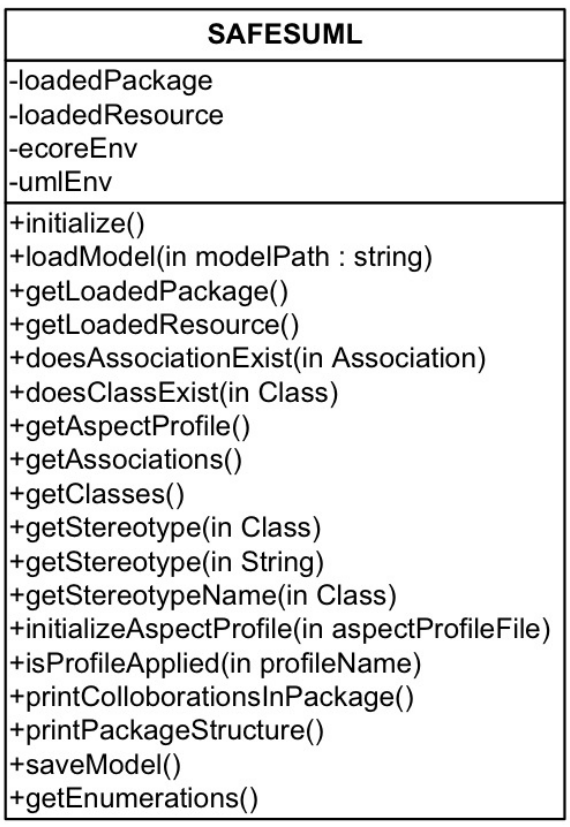


Figure 4.31: UML Class for SAFESUML

public boolean initialize(): handles all initialization routines for the UML environment. It loads necessary UML packages such as primitive types, UML metamodel, etc. into the environment before the actual UML model is loaded.

public boolean loadModel(String model): takes the file path of the XMI file containing the UML model as input, parses the file and loads the model into the environment together with its applied profiles.

public Package getLoadedPackage(): returns the package object after the actual model is loaded.

public Resource getLoadedResource(): returns the resource object after the actual model is loaded.

public Boolean doesAssociationExist(Association association1): given an association, checks existence of association in the design model.

public Boolean doesClassExist(Class class1): given a class, checks existence of class in the design model.

public Package getAspectProfile(): returns the UML profile associated with an aspect profile.

public List<Association> getAssociations(): returns the list of associations in the design model.

public List<Class> getClasses(): returns the list of classes in the design model.

public Stereotype getStereotype(Class cls): given a class, returns the applied stereotype

public Stereotype getStereotype(String name): given a stereotype name, returns the stereotype from the profile.

public String getStereotypeName(Class cls): given a class, returns the

name of the applied stereotype.

public boolean initializeAspectProfile(String aspectProfileFile): initializes given aspect profile in the UML environment.

public Boolean isProfileApplied(String profileName): given a profile name, checks if that profile already applied to the design model.

public void printPackageStructure(Package pkg): given a UML package, prints names of all elements in the package.

public void saveModel(): saves all the changes done on the loaded UML design model.

public List<Enumeration> getEnumerations(): returns the list of all enumerations in the loaded UML package.

4.4.4.2. SAFESOCL Library Class. SAFESOCL class (Figure 4.32) is a wrapper on top of Eclipse EMF OCL implementation. It simplifies the initialization of the OCL environment and provide simple methods to check constraints on a loaded UML model.

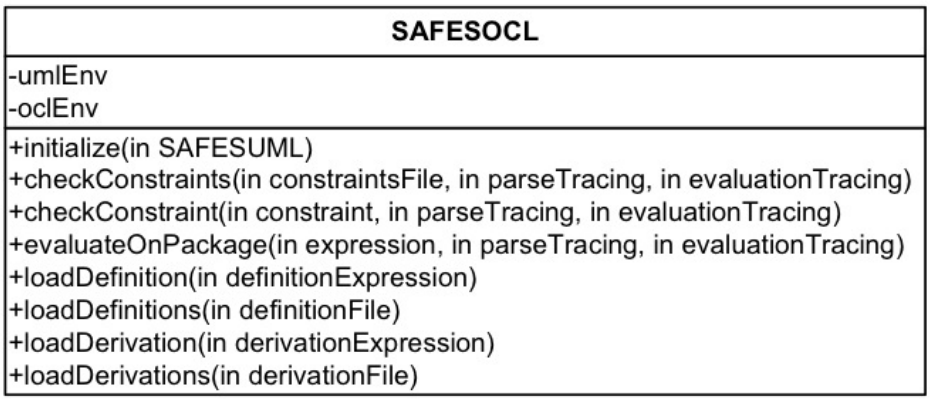


Figure 4.32: UML Class for SAFESOCL

public boolean initialize(SAFESUML uml): takes SAFESUML as input and handles all initialization routines for the OCL environment on top of the initialized UML environment.

public boolean checkConstraints(String constraintsFile, Boolean parseTracing, Boolean evaluationTracing): takes a text file containing OCL constraints as input, executes the constraints on the UML model loaded to the environment, and returns the boolean result. "parseTracing" and "evaluationTracing" parameters are provided for debugging purposes.

public boolean checkConstraint(String constraint, Boolean parseTracing, Boolean evaluationTracing): takes a string containing a single OCL constraint as input, executes the constraints on the UML model loaded to the environment, and returns the boolean result. "parseTracing" and "evaluationTracing" parameters are provided for debugging purposes.

public Object evaluateOnPackage(String exprString, Boolean parseTracing, Boolean evaluationTracing): takes a string containing a single OCL query expression as input, executes the expression on the UML model loaded to the environment, and returns the object result. "parseTracing" and "evaluationTracing" parameters are provided for debugging purposes.

public List<OCLExpression<Classifier>> loadDefinition(String definition): given an OCL definition as String input, loads the definition into the OCL environment.

public List<OCLExpression<Classifier>> loadDerivations(String derivationsFile): given the name of a file containing a list of OCL definitions, loads the definitions into the OCL environment.

public List<OCLExpression<Classifier>> loadDerivation(String derivation): given an OCL derivation as String input, loads the derivation into the OCL

environment.

public List<OCLExpression<Classifier>> loadDerivations(String derivationsFile)): given the name of a file containing a list of OCL derivations, loads the derivations into the OCL environment.

4.4.4.3. Security Function Merger Library Class. SecurityAspectMerger class (Figure 4.33) provides necessary functions to security function merger tool.

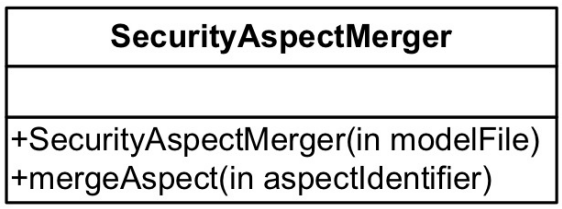


Figure 4.33: UML Class for Security Aspect Merger

public SecurityAspectMerger(String modelFile): given the name of the file containing the design model, this function initializes the UML and OCL environments for further processing.

public Boolean mergeAspect(String aspectIdentifier): given an aspect identifier, merges design model of the aspect to the loaded UML design model.

4.4.4.4. PROMELA Generator Library Class. UML2PROMELATransformer class (Figure 4.34) provides necessary functions to transform a UML design model (class diagram and state machine diagram) to a PROMELA model to be checked within SPIN model checker.

public UML2PROMELATransformer(): constructor method for UML2PROMELATran Initializes PROMELA specification object for further use.

public Spec uml2PromelaTransformation(List<Class> classes): main

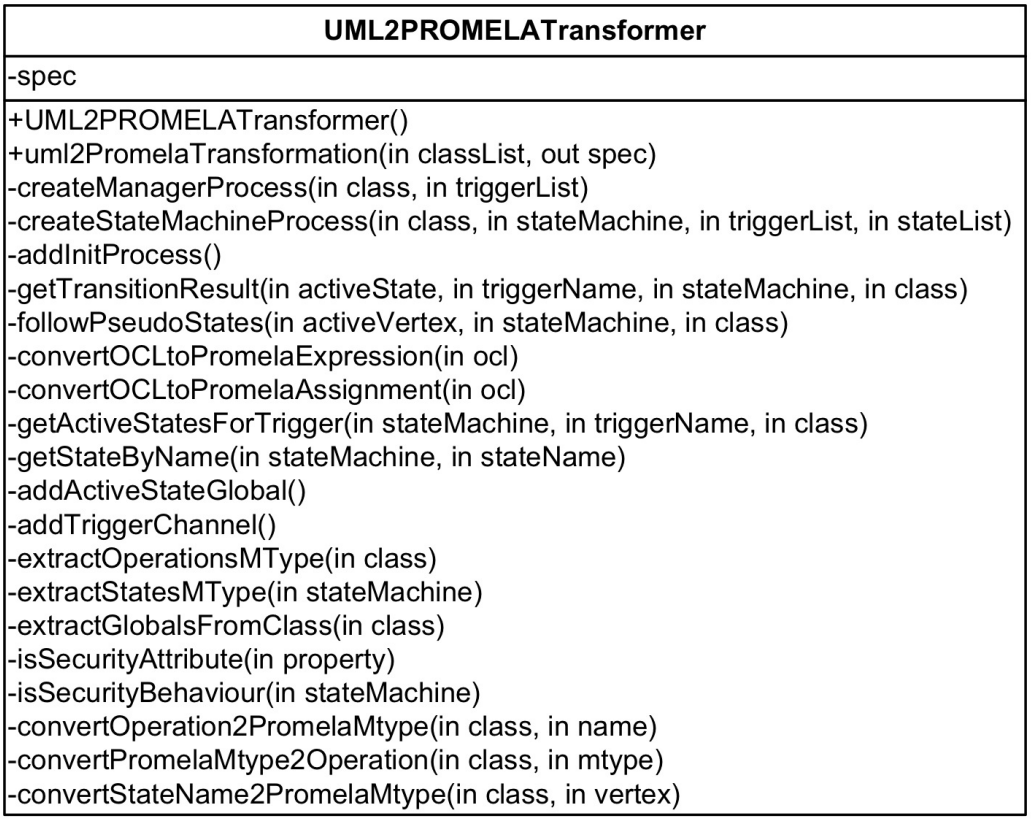


Figure 4.34: UML Class for UML to PROMELA Model Transformation

operation of the transformation engine. Given a list of classes from a UML class diagram, transforms both the static structure and dynamic behavior of the classes into a PROMELA model.

private void createManagerProcess(Class class1, List<String> triggers): given a class and a list of triggers, creates the main manager process of the PROMELA model. Main manager process is responsible to fire triggers to other processes.

private void createStateMachineProcess(Class class1, StateMachine sm, List<String> triggers, List<String> states): given a class, a state machine belonging to that class, list of triggers and list of states, creates a process reflecting the behavior presented in the state machine.

private void addInitProcess(): creates the init process for a PROMELA model.

private List<Stmnt> getTransitionResult(String activeState, String triggerName, StateMachine sm, Class class1):

private Stmnt followPseudoStates(Vertex activeVertex, StateMachine sm, Class class1): given a vertex in the state machine, checks transitions and choice pseudo states and identifies the target in a recursive manner and creates necessary conditional PROMELA statements for all the transition guards written in OCL.

private Expr convertOCLtoPromelaExpression(String ocl): Given a guard statement written in OCL, converts it to PROMELA expression.

private Stmnt convertOCLtoPromelaAssignment(String ocl): Given an assignment statement written in OCL, converts it to a PROMELA assign statement.

private List<String> getActiveStatesForTrigger(StateMachine sm, String triggerName, Class class1):

private Vertex getStateByName(StateMachine sm, String stateName): given a state machine and name of a state, finds the state.

private void addActiveStateGlobal(): adds a global variable into the PROMELA model to store active state.

private void addTriggerChannel(): adds a global variable into the PREMELA model to store trigger channel.

private List<String> extractOperationsMType(Class class1): given a class, converts all operations of the class to PROMELA mtype.

private List<String> extractStatesMType(StateMachine sm): given a state machine converts all states, pseudo states and final states of the state machine to PROMELA mtype.

private void extractGlobalsFromClass(Class class1): given a class, converts all attributes to PROMELA variables.

private Boolean isSecurityAttribute(Property property): given a UML property, checks if it is a security property.

private Boolean isSecurityBehaviour(StateMachine sm): given a state machine, checks if it is a security behaviour.

private String convertOperation2PromelaMtype(Class class1, String name): given a class and an operation, converts operation name to PROMELA mtype.

private String convertPromelaMtype2Operation(Class class1, String mtype): given a class and a PROMELA mtype, converts mtype to corresponding class operation.

private String convertStateName2PromelaMtype(Class class1,Vertex vertex): given a class and a vertex, converts name of the vertex to PROMELA mtype.

4.4.4.5. Persistence Layer Classes. Toolset contains a persistence layer, which is responsible to provide data layer support for the processes in the process catalog. Persistence layer is based on HyperSQL Database (HSQLDB), which is a database engine written in java. The reason for using HSQLDB is to remove the need to install a separate database server to be used with the framework. With HSQLDB it's possible to bundle a standalone (no-install) database to be used within the framework distribution. Scheme of the embedded database is described in Figure 4.35.

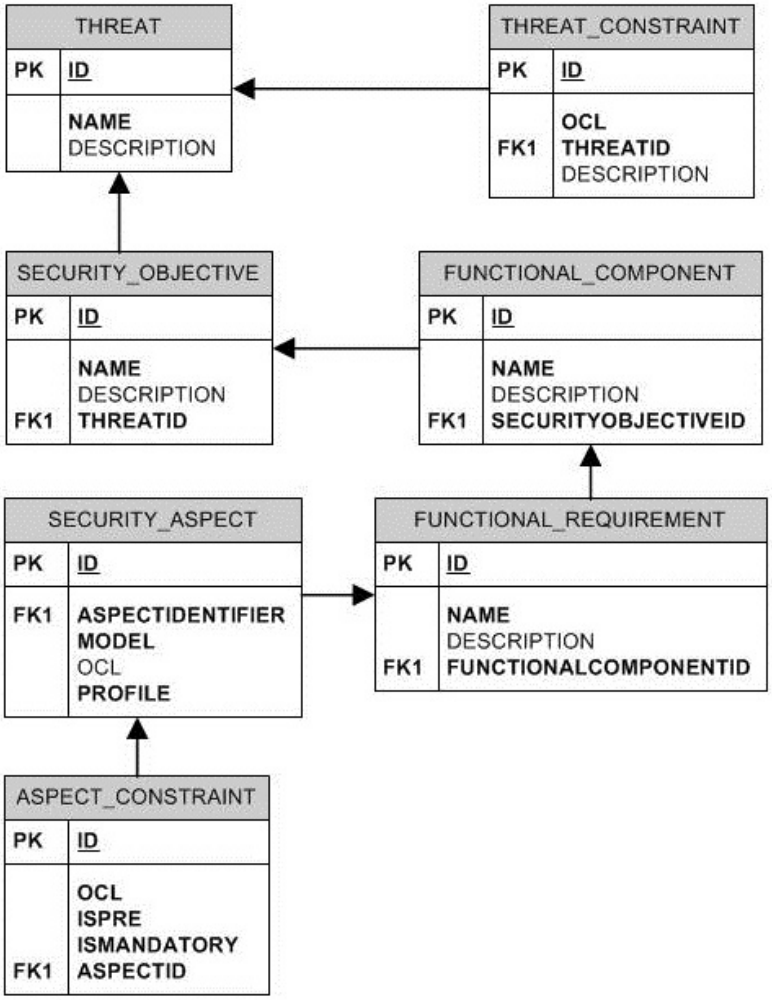


Figure 4.35: Database Schema for Persistence Layer

Threat library is stored in Threat table. Only the names and descriptions of the threats are maintained in the database. As described in Threat Auto Detection process in Section 4.4.3.2, threats are identified using OCL constraints. There may be several constraints, which help to identify the same threat. Constraints are stored in the file system, within the framework folder structure and paths of constraints are stored in Threat_Constraint database table. According to the Security Objective Decomposition process (Section 4.4.3.3), threats are decomposed into the related security objectives. Security objectives, their descriptions and relations to threat are stored in Security_Objective table. Every security objective is further decomposed into Functional Components and Security Functional Requirements of the Common Criteria (Section 4.4.3.8). Related functional components and security functional requirements are stored in the FunctionalComponent and Functional_Requirement tables.

For some of the security functional requirements, MDSF library contains predesigned security aspects, which can be automatically embedded to the design model of the software. Definitions of these aspects are stored in Security_Aspect table. Every defined security aspect has some associated constraints stored in Aspect_Constraint. These contains allow verifying the design model before and after the aspect design is merged into the design model of the software. Using these constraints provides a way to overcome model inconsistencies and incorrect merge operations.

ThreatAnalysisPersistence:

ThreatAnalysisPersistence class provides necessary functions to query threats and threat identification constraints from the threat library. Operations provided by the class are shown in Figure 4.36. “getThreats” operation returns a list of all threats in the threat library. “getThreatsWithConstraints” operation returns the list of threats together with all the OCL constraints which are used to identify the threats. “shutdown” operation terminates the persistence session and closes all the database connections.

SecurityObjectiveDecompositionPersistence:

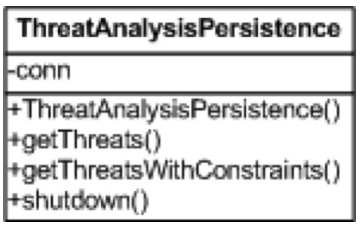


Figure 4.36: UML Class for ThreatAnalysisPersistence

SecuritObjectiveDecompositionPersistence class provides necessary functions to query security objectives associated with a threat. Operations provided by the class are shown in Figure 4.37 "getSecurityObjectives" operation returns the list of security objectives from the Security Objective and SFR Decomposition Library (Section 4.3.6) for a specified threat. "shutdown" operation terminates the persistence session and closes all the database connections.

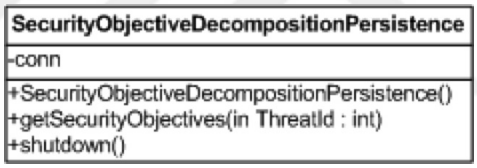


Figure 4.37: UML Class for SecurityObjectiveDecompositionPersistence

SFRDecompositionPersistence

SFRDEcompositionPersistence class provides necessary functions to query security functional requirements associated with a security objective. "getFunctionalComponents" operation returns the list of Common Criteria functional components for a given security objective. "getFunctionalComponentsWithRequirements" operation returns the list of CommonCriteria functional components together with corresponding security functional requirements for a given security objective as described in Section 4.3.4 and Section 4.4.3.8. "getFunctionalRequirements" operation returns the list of security functional requirements for a given function component. "shutdown" operation terminates the persistence session and closes all the database connections.

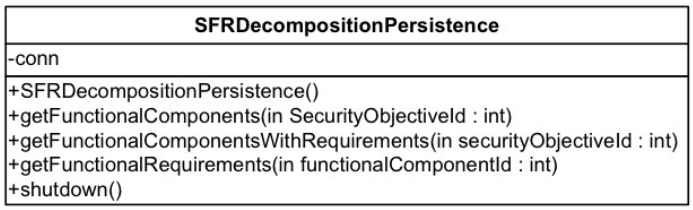


Figure 4.38: UML Class for SFRDecompositionPersistence

SecurityAspectPersistence

SecurityAspectPersistence class provides necessary functions to query security aspects and associated security aspect constraints. "getSecurityAspect" operation takes aspectIdentifier or aspectId as input and returns the associated security aspect from the library. "getSecurityAspectWithConstraints" operation takes aspectIdentifier or aspectId as input and returns the associated security aspect from the library together with all OCL constraints for the security aspect. "getSecurityAspectConstraints" operation takes id of a security aspect and returns all OCL constraints associated with the security aspect. "shutdown" operation terminates the persistence session and closes all the database connections.

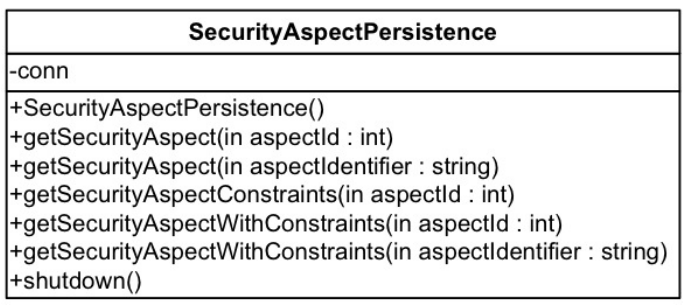


Figure 4.39: UML Class for SecurityAspectPersistence

4.4.4.6. MDSF Model Classes. Model package contains all necessary classes, which are used by persistence layer classes:

- Threat
- Constraint
- SecurityObjective
- FunctionalComponent
- FunctionalRequirement
- SecurityAspect
- SecurityAspectConstraint

Details of these classes are directly aligned with the database schema displayed in Figure 4.35.

4.5. Novelties of MDSF for Software Analysis, Design and Verification

Currently, stakeholders in the software development cycle need to learn and use different languages, tools and methodologies for different security problems. This is very infeasible for the software industry because of complexity, time and cost constraints. Software industry need a common language and a combination of powerful methods and tools to analyze, design, and verify the security features of information systems. UML is the industry-standard for analyzing and designing the artifacts of an information system. However, expressive power and formalism of barebone UML are not enough for security analysis, design, and verification. On the other hand, most of the open-source and commercial modeling tools support UML profiles as a standard extension mechanism. In MDSF, we propose a new UML profile as an extension for annotating the analysis and design models to formalize the security requirements and security functions within the models. Several proposal, including but not limited to UMLSec [18] [19] [20], SECDW [23], SecureUML [26], For-LySa [34], and SECTET [36] [37] define a new UML Profile for security analysis and design. Differently, in MDSF we also align our new UML profile with the Common Criteria to support the certification process and complement it with tools for security verification.

All of the methods explained in the Literature Survey section consider security from a different angle and do not address a wider range of security problems. Some of

them focus only on access control and authentication, some focus on web application security, and still others focus on data warehouses or SOA. In MDSF, our aim is to provide an extendable framework with a common language and a set of tools for covering analysis, design and verification for all of the security requirements defined in the Common Criteria. Currently, with MDSF UML profile and tools we support authentication, authorization, information flow, and audit security families of the Common Criteria. MDSF is designed as extendable to support the rest of the security domains.

UMLSec [18] [19] [20] and some other proposals [39] [43] [45] [47] explained how they can use model checking for verification of the security functionality in the information system designs. MDSF also uses model checking and MDD for verification of security properties on the UML analysis and design models. The existing stereotypes in UMLSec and other UML profiles are not sufficient to support the Common Criteria requirements. Therefore, in MDSF, we have proposed a new UML profile to address these challenges and defined a model driven transformation approach to convert the UML models to PROMELA for model checking in Spin Model Checker. Differently, in addition to the new UML profile, MDSF also uses OCL and LTL to increase the expressiveness and formalization of the barebone UML.

Some researchers [53] [10] [54] worked on a security analysis methodology aligned with the Common Criteria. We agree that their approach is logical and needed for aligning the security analysis and design phases with the Common Criteria framework. In MDSF, we employ a similar methodology to identify possible security threats using UML models, map these threats to security objectives from the Common Criteria and decompose them to security requirements for the system design. Differently in MDSF, we not only make use of the UML use case diagrams, but we also use UML sequence diagrams to analyze and identify threats. In addition, instead of using proprietary mechanisms for threat identification and security requirement extraction from the UML models, MDSF uses standard UML profile mechanisms and OCL invariants. Lastly, in MDSF, we have also used the reusable security function design concept [55] and designed a tool for merging security function designs into the design model of the information system.

In summary, major contributions of MDSF are as follows:

- A framework with necessary tools to support the analysis and design of security features of an IT product and ease the Common Criteria certification process with formal verification of security policies on the design model.
- A tool and a new UML profile (MDSF Threat Analysis Profile) supported with OCL invariants for automated threat analysis and identification
- A tool for extraction of security objectives and security functional requirements from the identified security threats
- New UML profiles to design for identification and authentication, access control, audit and information flow security requirements in the Common Criteria
- A tool for merging already available security function designs into a design model
- OCL queries and invariants to check the design model for completeness, conformance and security requirements
- A tool for transforming the UML design models into PROMELA specifications to be checked in the Spin model checker.

5. PROOF OF CONCEPT CASE STUDIES

5.1. Brief Explanation of the Presented Proof of Concepts

In this thesis, seven different proof of concepts are presented to explain different steps and tools of the framework:

- Proof of Concept 1: Explains the MDSF Threat Analysis Profile and Threat Identification Tool and their application on an example model.
- Proof of Concept 2: Explains the security function repository and Security Function Merger Tool and their application on an example model.
- Proof of Concept 3: Explains the MDSF Identification and Authentication Profile, PROMELA Generator Tool and verification of security requirements via model checking.
- Proof of Concept 4: Explains the MDSF Audit Profile, OCL invariants for static requirement verification, PROMELA Generator Tool and verification of security requirements via model checking.
- Proof of Concept 5: Explains the MDSF Information Flow Profile, PROMELA Generator Tool and verification of security requirements via model checking.
- Proof of Concept 6: Explains the embedded formal specifications in the design model and how they are converted to PROMELA and verified using model checking
- Proof of Concept 7: An end to end example is provided to explain all of the MDSF tools and processes in action.

5.2. Proof of Concept 1: Extraction of Security Requirements from an Analysis Model

5.2.1. Rationale

The aim of this proof of concept is to show, given a simple analysis model, how framework assists to extract possible threats and security functional requirements.

5.2.2. Description

In this proof of concept, a simple service delivery platform architecture composed of several software components is being analyzed. This service delivery platform is responsible to serve several clients. Service requests are received by the service delivery platform, delivered to appropriate services; responses are collected and send back to the requesting clients. Client and services are assumed to be out of system boundaries.

5.2.3. Inputs

An analysis model consisting of a high-level class diagram Figure 5.1 and a use case description given in a sequence diagram Figure 5.2 is provided as input to this use case.

In this basic proof of concept, a user can utilize several types of clients to access the services delivered by the platform. Every client is connected to the service delivery platform with a different connection mechanism.

Services can be of several types such as IMS Service, IN Service or Content Service. Each service has specific service information, which is stored in the service catalog. Service catalog is accessed by the service delivery platform.

Every user has attached user information containing user preferences, credentials and active subscriptions that are stored in a user profile database. Platform access this

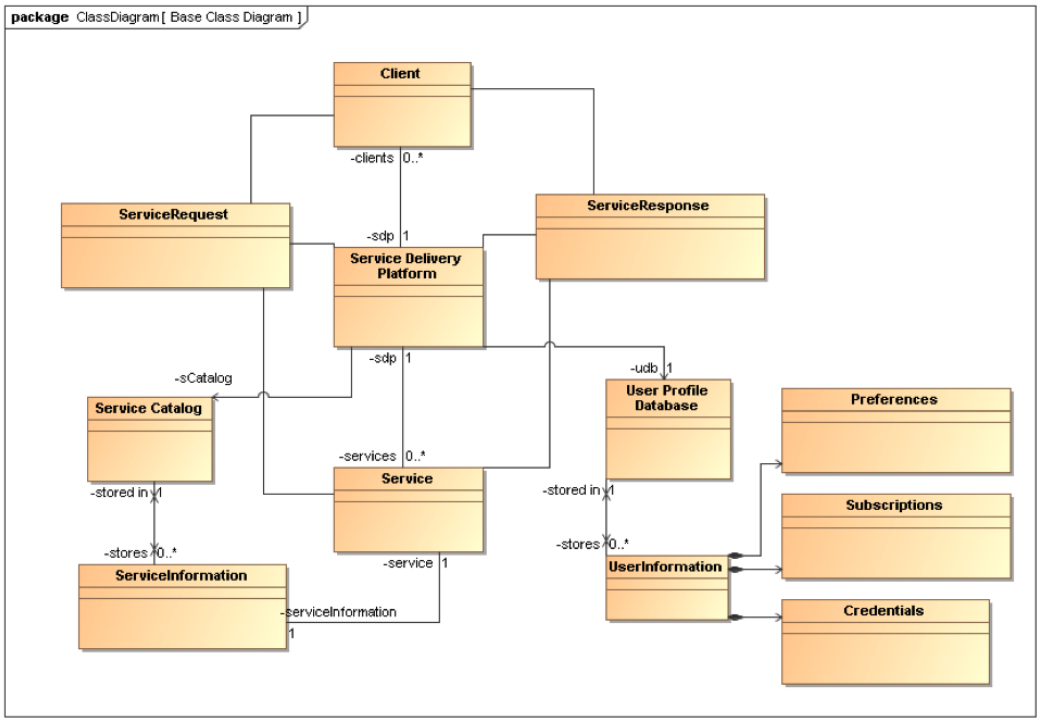


Figure 5.1: Sample System High Level Class Diagram

user database to get and update user information.

In this high-level sequence diagram, a simple use case is described. In this scenario, Client sends Credentials to Service Delivery Platform. Service Delivery Platform checks for the credentials using the User Profile Database. Afterwards, Client sends ServiceRequest to the Service Delivery Platform. Service Delivery Platform identifies the service by sending ServiceRequest to the Service Catalog. Service Delivery Platform handles charging of the request by sending the ServiceRequest to the Convergent Charging System. After successful charging, Service Delivery Platform sends ServiceRequest to the appropriate Service. Service executes the request and returns a ServiceResponse. Service Delivery Platform sends this ServiceResponse back to the Client.

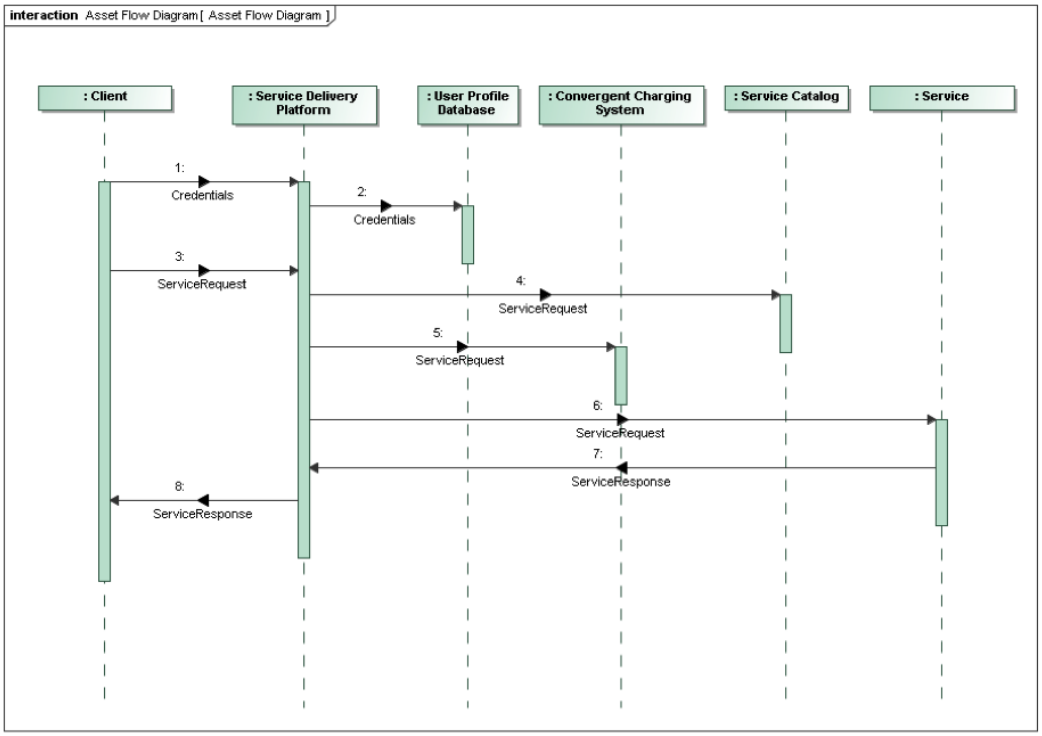


Figure 5.2: Use Case Description - UML Sequence Diagram

5.2.4. Expected Outputs

Outputs of this use case are a list of possible threats which may threaten the system and a list of security functional requirements, when satisfied, that will mitigate these threats.

5.2.5. Execution Steps

5.2.5.1. Step 1 - Critical Asset Identification. On the analysis model's class diagram, critical assets are marked with "Asset" stereotype from the MDSF UML Profile.

In Figure 5.3, all critical assets are marked in the class diagram for the proof of concept:

- Credentials (UserInformation): Credentials class is marked as HighImpact in the

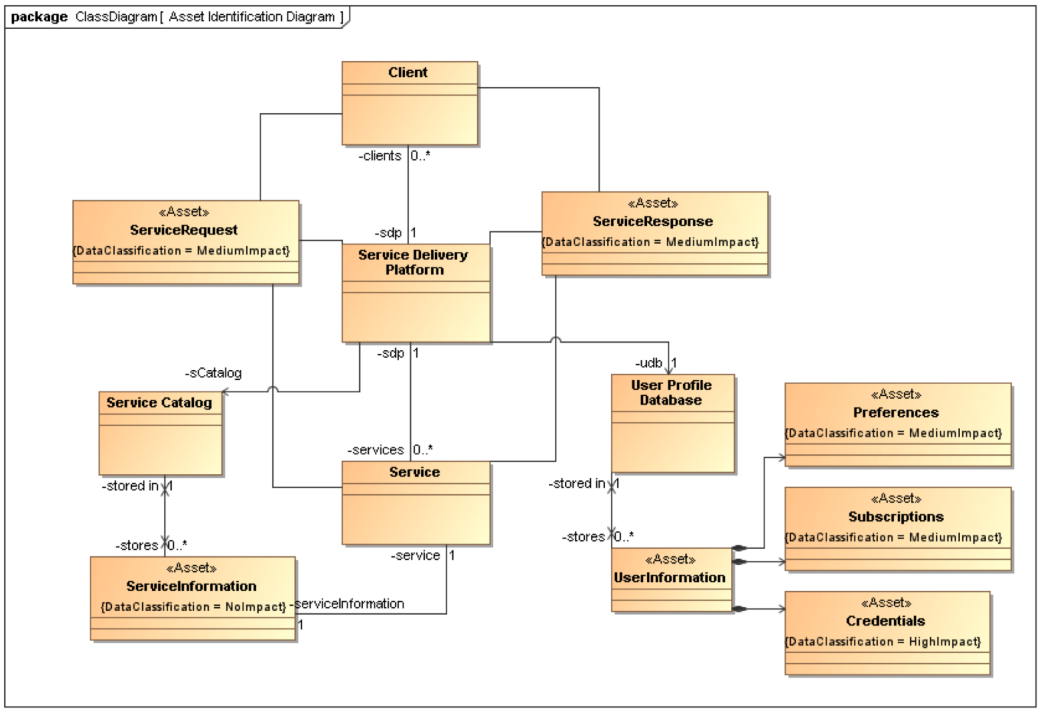


Figure 5.3: Critical Asset Identification

DataClassification tagged value. The reason for high impact is straightforward since disclosure of credentials will result in serious security issues in the system and may cause financial loss for the user.

- Preferences (UserInformation): Preferences is marked as MediumImpact since leakage of preferences will not directly cause financial loss for the user.
- Subscriptions (UserInformation): Subscriptions is marked as MediumImpact since leakage of subscriptions will not directly cause financial loss for the user.
- ServiceRequest: ServiceRequest is marked as MediumImpact since leakage of service request information will not directly cause financial loss for the user.
- ServiceResponse: ServiceResponse is marked as MediumImpact since leakage of service response information will not directly cause financial loss for the user.
- ServiceInformation: ServiceInformation is marked as NoImpact since leakage of service information will not cause any trouble to the system and users.

5.2.5.2. Step 2 - Threat Auto Detection. In MDSF UML Profile every UML element can have TrustLevel tagged value. This tagged value is used to determine possible existence of information disclosure threat. TrustLevel tagged value contains High, Medium, Low and Unknown values. In order not to have a complicated class diagram during analysis, a subset of the class diagram is taken and TrustLevel tagged values are assigned as shown in Figure 5.4. In this subset, only the classes, which are directly involved in the communication, are taken. This is why this diagram is named as the Communication Diagram.

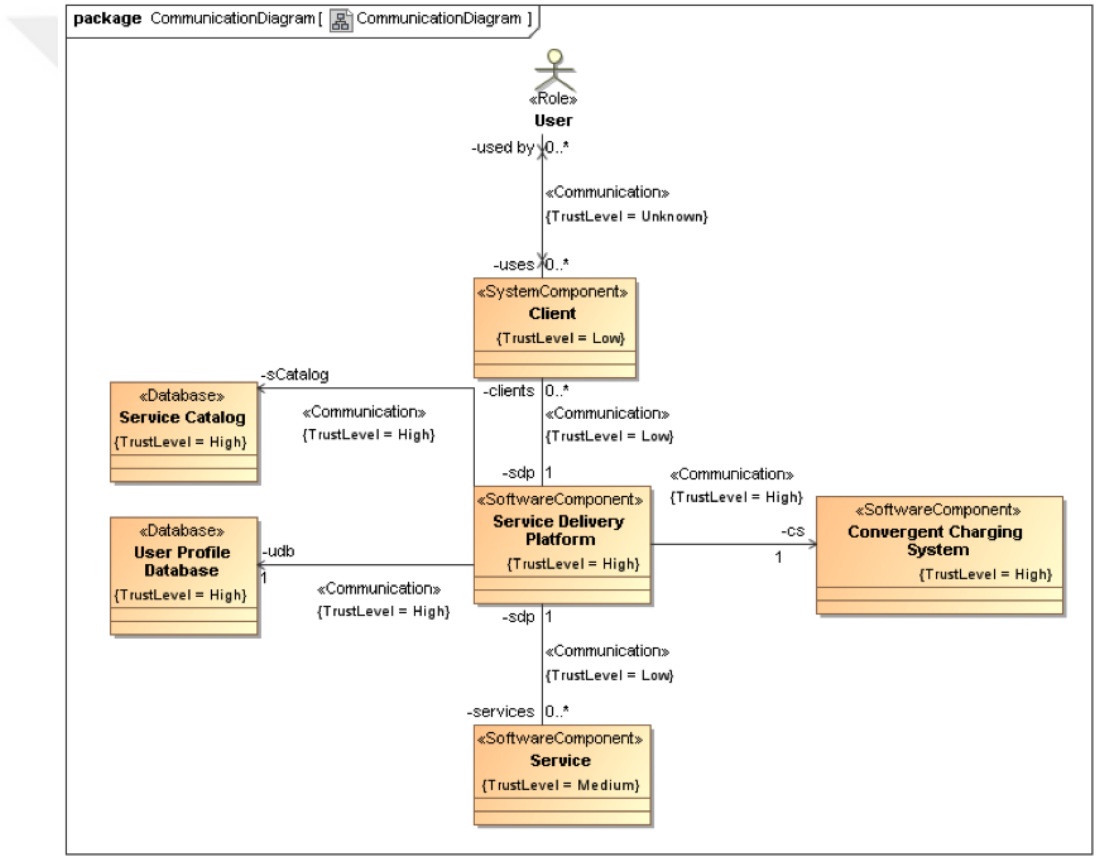


Figure 5.4: Communication Diagram

Database, SoftwareComponent and SystemComponent stereotypes are also assigned to the entities where appropriate. User Profile Database and Service Catalog are assigned “Database” stereotype. Service Delivery Platform, Service and Convergent Charging System are assigned “SoftwareComponent” stereotype. Client entity is assigned “SystemComponent” stereotype and User is assigned “Role” stereotype. All

of these stereotypes have some different tagged values (Section 4.2.1), which will help automatic threat analysis to identify possible threats. For this specific proof of concept, only needed tagged value is TrustLevel, which is common to all MDSF stereotypes.

In order to identify the communication properties between these entities, "Communication" stereotype of MDSF UML Profile is assigned to the Associations between these classes. TrustLevel of the Communication lines are also specified using the TrustLevel tagged values. In Figure 5.4, TrustLevel of communication between User and Client is marked as unknown since analysis will start with the Client. TrustLevel of the Client is defined as "Low" since from the system point of view, Client is a remote unknown entity. TrustLevel of the Service is "Medium" since services are not deployed in the system boundaries and offered by 3rd party partners. All other entities have TrustLevel as "High" because they are in the system boundaries. Communication between Client and Service Delivery Platform, Service and Service Delivery Platform are marked as "Low" because the lines are not controlled by the system itself.

After execution of threat identification OCL constraints on the diagrams, TIT identifies existence of T.Disclose_Data threat. According to ECMA Protection Profile, E-COFC Public Business Class [45], information may be disclosed in to unauthorized users. This includes user information, system information and business data. Information may be disclosed while being stored or processed in the TOE, or during transmission. Disclosure of authentication data during transmission would allow someone to logon and assume the identity of an authorized user.

5.2.5.3. Step 3 - Security Objective Decomposition . At this step T.Disclose_Data threat identified at Step 2 will be decomposed to some security objectives in order to have a mitigation scenario. MDSF Library contains a threat repository, security objective repository and their relations as described in Section 4.3. Based on these repositories and defined relationships, security objectives defined in Table 5.1 are identified automatically by the toolset.

Table 5.1: Identified Security Objectives

Security Objective	**Description**
O.Access_Control	The TOE must enforce an access control policy protects information from unauthorized access.
O.Authen	Users will be uniquely identified and authenticated prior to performing any actions to be mediated by the TOE.
O.Flow_Control	The TOE must enforce an information flow control policy at specified boundaries.
O.Resid_Prot	Information stored within the TOE must not be made available to other users upon release.
O.Authen_Protect	The TOE must protect authentication information.

5.2.5.4. Step 4 - SFR Decomposition. Security objectives identified at Step 3 are further decomposed into security functional requirements at this step. MDSF Library contains necessary decomposition rules in the repositories. Based on these rules, security functional components defined in Table 5.2 are identified automatically by the toolset.

Table 5.2: Identified Security Functional Components

Number	**Title**	**Objective**
FDP_ACC.1	Subset access control	O.Access_Control
FDP_ACF.1	Security attribute based access control	O.Access_Control
FMT_SMR.2	Restrictions on security roles	O.Access_Control
FAU_SAR.2	Restricted audit review	O.Access_Control
FMT_MSA.3	Static attribute initialization	O.Access_Control
FPR_UNO.1	Unobservability	O.Access_Control
FIA_UID.1	Timing of identification	O.Authen

FIA_UAU.1	Timing of authentication	O.Authen
FAU_GEN.1	Audit data generation	O.Authen
FIA_UAU.3	Unforgeable authentication	O.Authen
FIA_ATD.1	User attribute definition	O.Authen
FMT_MSA.1	Management of security attributes	O.Authen
FTA_TAB.1	Default TOE access banners	O.Authen
FMT_MTD.1	Management of TSF data	O.Authen
FTA_TAH.1	TOE access history	O.Authen
FTA_SSL.1	TSF-initiated session locking	O.Authen
FTA_SSL.3	TSF-initiated termination	O.Authen
FIA_UAU.6	Re-authenticating	O.Authen
FTA_TSE.1	TOE session establishment	O.Authen
FIA_UAU.5	Multiple authentication mechanisms	O.Authen
FCS_CKM.2	Cryptographic key distribution	O.Authen
FDP_IFC.1	Subset information flow control	O.Flow_Control
FDP_IFF.1	Simple security attributes	O.Flow_Control
FDP_RIP.2	Full residual information protection	O.Resid_Prot
FCS_CKM.3	Cryptographic key access	O.Authen_Protect
FIA_SOS.1	Verification of secrets	O.Authen_Protect

These functional components are further decomposed into the implied security functional requirements given in Table 5.3.

Table 5.3: Identified Security Functional Requirements

SFR Number	**SFR Body**
FDP_ACC.1.1	The TSF shall enforce the [assignment: access control SFP] on [assignment: list of subjects, objects, and operations among subjects and objects covered by the SFP]
FDP_ACF.1.1	The TSF shall enforce the [assignment: access control SFP] to objects based on the following: [assignment: list of subjects and objects controlled under the indicated SFP, and for each, the SFP-relevant security attributes, or named groups of SFP-relevant security attributes]
FDP_ACF.1.2	The TSF shall enforce the following rules to determine if an operation among controlled subjects and controlled objects is allowed: [assignment: rules governing access among controlled subjects and controlled objects using controlled operations on controlled objects]
FDP_ACF.1.3	The TSF shall explicitly authorize access of subjects to objects based on the following additional rules: [assignment: rules, based on security attributes, that explicitly authorize access of subjects to objects]
FDP_ACF.1.4	The TSF shall explicitly deny access of subjects to objects based on the following additional rules: [assignment: rules, based on security attributes, that explicitly deny access of subjects to objects]
FMT_SMR.2.1	The TSF shall maintain the roles: [assignment: authorized identified roles]

FMT_SMR.2.2	The TSF shall be able to associate users with roles
FMT_SMR.2.3	The TSF shall ensure that the conditions [assignment: conditions for the different roles] are satisfied
FAU_SAR.2.1	The TSF shall prohibit all users read access to the audit records, except those users that have been granted explicit read-access
FMT_MSA.3.1	The TSF shall enforce the [assignment: access control SFP, information flow control SFP] to provide [selection, choose one of: restrictive, permissive, [assignment: other property]] default values for security attributes that are used to enforce the SFP
FMT_MSA.3.2	The TSF shall allow the [assignment: the authorized identified roles] to specify alternative initial values to override the default values when an object or information is created
FPR_UNO.1.1	The TSF shall ensure that [assignment: list of users and/or subjects] are unable to observe the operation [assignment: list of operations] on [assignment: list of objects] by [assignment: list of protected users and/or subjects]
FIA_UID.1.1	The TSF shall allow [assignment: list of TSF-mediated actions] on behalf of the user to be performed before the user is identified
FIA_UID.1.2	The TSF shall require each user to be successfully identified before allowing any other TSF-mediated actions on behalf of that user

FIA_UAU.1.1	The TSF shall allow [assignment: list of TSF mediated actions] on behalf of the user to be performed before the user is authenticated
FIA_UAU.1.2	The TSF shall require each user to be successfully authenticated before allowing any other TSF-mediated actions on behalf of that user
FAU_GEN.1.1	The TSF shall be able to generate an audit record of the following auditable events: a) Start-up and shutdown of the audit functions; b) All auditable events for the [selection, choose one of: minimum, basic, detailed, not specified] level of audit; and c) [assignment: other specifically defined auditable events]
FAU_GEN.1.2	The TSF shall record within each audit record at least the following information: a) Date and time of the event, type of event, subject identity (if applicable), and the outcome (success or failure) of the event; and b) For each audit event type, based on the auditable event definitions of the functional components included in the PP/ST, [assignment: other audit relevant information]
FIA_UAU.3.1	The TSF shall [selection: detect, prevent] use of authentication data that has been forged by any user of the TSF
FIA_UAU.3.2	The TSF shall [selection: detect, prevent] use of authentication data that has been copied from any other user of the TSF

FIA_ATD.1.1	The TSF shall maintain the following list of security attributes belonging to individual users: [assignment: list of security attributes]
FMT_MSA.1.1	The TSF shall enforce the [assignment: access control SFP(s), information flow control SFP(s)] to restrict the ability to [selection: change_default, query, modify, delete, [assignment: other operations]] the security attributes [assignment: list of security attributes] to [assignment: the authorized identified roles]
FTA_TAB.1.1	Before establishing a user session, the TSF shall display an advisory warning message regarding unauthorized use of the TOE
FMT_MTD.1.1	The TSF shall restrict the ability to [selection: change_default, query, modify, delete, clear, [assignment: other operations]] the [assignment: list of TSF data] to [assignment: the authorized identified roles]
FTA_TAH.1.1	Upon successful session establishment, the TSF shall display the [selection: date, time, method, location] of the last successful session establishment to the user
FTA_TAH.1.2	Upon successful session establishment, the TSF shall display the [selection: date, time, method, location] of the last unsuccessful attempt to session establishment and the number of unsuccessful attempts since the last successful session establishment

FTA_TAH.1.3	The TSF shall not erase the access history information from the user interface without giving the user an opportunity to review the information
FTA_SSL.1.1	The TSF shall lock an interactive session after [assignment: time interval of user inactivity] by: a) clearing or overwriting display devices, making the current contents unreadable; b) disabling any activity of the user's data access/display devices other than unlocking the session
FTA_SSL.1.2	The TSF shall require the following events to occur prior to unlocking the session: [assignment: events to occur]
FTA_SSL.3.1	The TSF shall terminate an interactive session after a [assignment: time interval of user inactivity]
FIA_UAU.6.1	The TSF shall re-authenticate the user under the conditions [assignment: list of conditions under which re-authentication is required]
FTA_TSE.1.1	The TSF shall be able to deny session establishment based on [assignment: attributes]
FIA_UAU.5.1	The TSF shall provide [assignment: list of multiple authentication mechanisms] to support user authentication
FIA_UAU.5.2	The TSF shall authenticate any user's claimed identity according to the [assignment: rules describing how the multiple authentication mechanisms provide authentication]

FDP_IFC.1.1	The TSF shall enforce the [assignment: information flow control SFP] on [assignment: list of subjects, information, and operations that cause controlled information to flow to and from controlled subjects covered by the SFP]
FDP_IFF.1.1	The TSF shall enforce the [assignment: information flow control SFP] based on the following types of subject and information security attributes: [assignment: list of subjects and information controlled under the indicated SFP, and for each, the security attributes]
FDP_IFF.1.2	The TSF shall permit an information flow between a controlled subject and controlled information via a controlled operation if the following rules hold: [assignment: for each operation, the security attribute-based relationship that must hold between subject and information security attributes]
FDP_IFF.1.3	The TSF shall enforce the [assignment: additional information flow control SFP rules]
FDP_IFF.1.4	The TSF shall explicitly authorize an information flow based on the following rules: [assignment: rules, based on security attributes, that explicitly authorize information flows]
FDP_IFF.1.5	The TSF shall explicitly deny an information flow based on the following rules: [assignment: rules, based on security attributes, that explicitly deny information flows]

FDP_RIP.2.1	The TSF shall ensure that any previous information content of a resource is made unavailable upon the [selection: allocation of the resource to, deallocation of the resource from] all objects
FCS_CKM.3.1	The TSF shall perform [assignment: type of cryptographic key access] in accordance with a specified cryptographic key access method [assignment: cryptographic key access method] that meets the following: [assignment: list of standards]
FIA_SOS.1.1	The TSF shall provide a mechanism to verify that secrets meet [assignment: a defined quality metric]

5.3. Proof of Concept 2: Weaving Security Aspects into Design Model

5.3.1. Rationale

The aim of this proof of concept is to show, given a simple design model and a single security requirement, how framework assists to merge a security aspect from the library into the design.

5.3.2. Description

In this proof of concept a simple software system is being designed for a company. This software will be used to manage trainings and training applications of employees within the company. In a simple scenario, employees enroll for the trainings. Manager of the department which employee works in approve these enrollments and instructors provide the training to the employees.

In this proof of concept we assume that after several analyses it's found out that FIA_ATD.1.1 security functional requirement is necessary for this software:

FIA_ATD.1.1: The TSF shall maintain the following list of security attributes belonging to individual users: [assignment: list of security attributes].

For more information on this requirement please refer to Section 4.3.2.1

MDSF Library contains a security aspect corresponding to this requirement. In the proof of concept, it'll be shown that using the provided framework toolset, it's possible to automatically weave this security aspect into the design model, which will satisfy the security functional requirement.

5.3.3. Inputs

A UML class diagram for the high level design of software components is provided as input to the proof of concept (Figure 5.5).

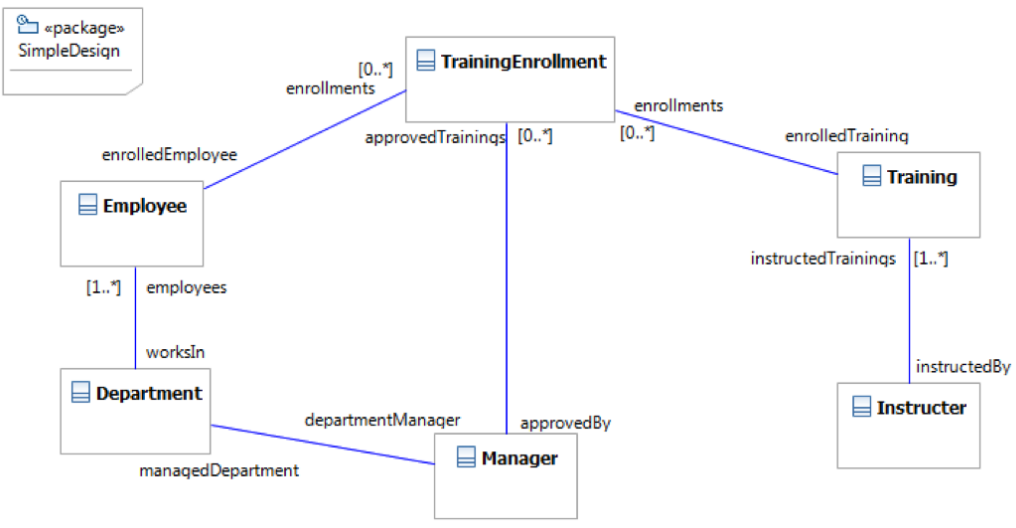


Figure 5.5: UML Class Diagram for Security Aspect Weaving

This class diagram provides a static view on the classes and relations within the software, which will be developed.

5.3.4. Expected Outputs

Expected output of the proof of concept is a merged UML class diagram, which also contains the classes, generalizations, relations and enumerations from the security aspect (Section 4.3.2.1).

5.3.5. Execution Steps

5.3.5.1. Step 1 - Stereotyping the Design Model. In order to identify the merge points for the security aspects, it's necessary to annotate design model with stereotypes from the MDSF UML Profile. For this proof of concept, security aspect for FIA_ATD.1.1 requirement will be weaved into the design model. This is why MDSF Identification and Authentication (FIA) UML Profile (Section 4.2.1) will be used for stereotyping the model.

Figure 5.6 shows the input design model stereotyped with Identification and Authentication UML Profile. Only three classes have stereotypes applied; Employee, Manager and Instructor. Applied stereotype is "UserExtension". This stereotypes show the security aspect merger tool correct places where and how to bind security aspect design into the model.

5.3.5.2. Step 2 - Executing Security Function Merger Tool. Design model together with stereotypes is input to the security function merger tool (Section 4.4.2.2). A security aspect from the library is selected within the tool. For this proof of concept, security aspect for FIA_ATD.1.1 is selected. After execution of the tool, an updated design diagram is created automatically as displayed in Figure 5.7.

Tool identifies the extension points in the model and uses those extension points as specific classes for a generalization association. For this proof of concept, extension points are the classes stereotyped with "UserExtension". After classes, associations and enumerations from the security aspect diagram are merged into the design model,

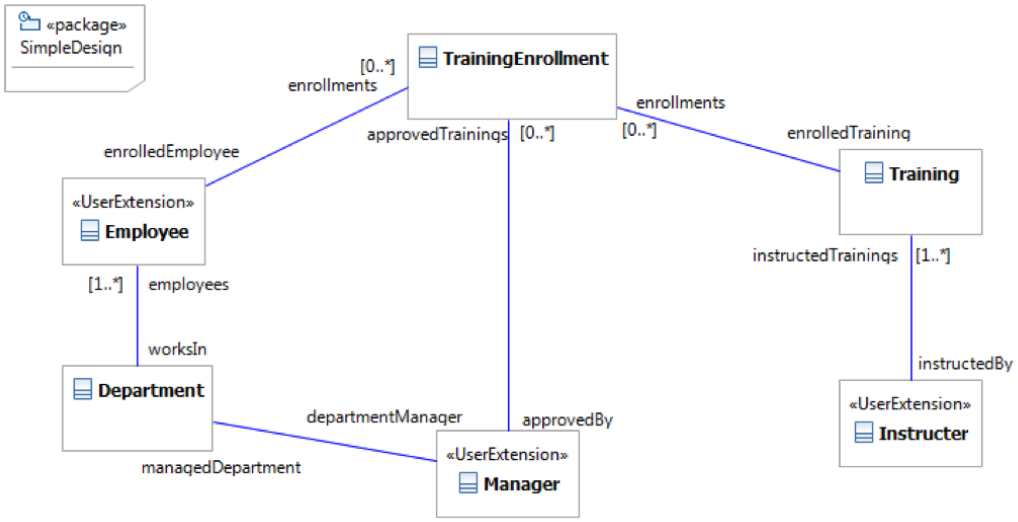


Figure 5.6: UML Class Diagram for Security Aspect Weaving (Stereotyped)

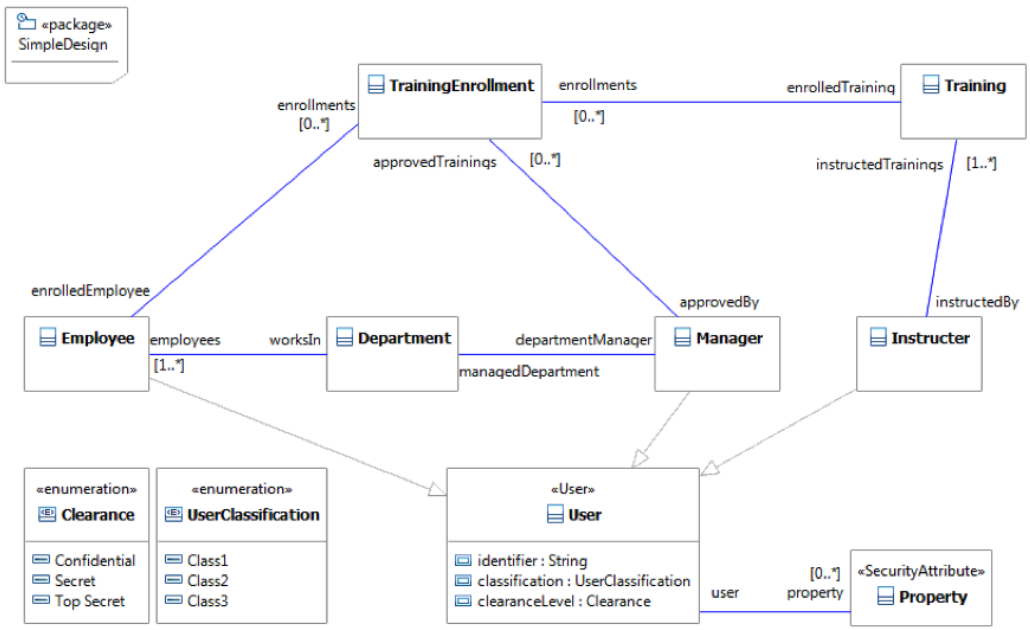


Figure 5.7: UML Class Diagram for Security Aspect Weaving (After Merge)

the tool creates necessary generalizations in order to connect aspect classes with the design classes as shown in Figure 5.7.

5.4. Proof of Concept 3: Model Checking Designs - Identification and Authentication

5.4.1. Rationale

Aim of this proof of concept is to show model transformation and model checking capabilities of MDSF.

5.4.2. Description

In this proof of concept, one of the designed security aspects of MDSF will be checked if it really satisfies the Common Criteria security functional requirement. Security aspect described in Section 4.3.2.2 will be taken as the input design. This security aspect is designed to satisfy FIA_AFL.1.1 and FIA_AFL.1.2 security functional requirements of the Common Criteria. This proof of concept assumes that design has passed static verification with execution of OCL invariants described in the corresponding section. As described in Section 4.4.3.10, model checking is used to verify requirements, which are related to dynamic behavior of the objects. For this design, the requirement we want to verify is given below:

req3 Whenever a user exceeds the maximum number of unsuccessful authentications, system shall detect it and lock the user account.

This requirement is directly related to the dynamic behavior of the class, which is responsible for handling login operations. This is why it's not possible to verify this property on a design model with execution of OCL invariants.

5.4.3. Inputs

Input to this proof of concept is a UML design, which contains security, related attributes and behaviors, which are expected to solve a security problem defined in a functional security requirement.

Design shall contain a UML class diagram (Figure 5.8) and one or more state machines (Figure 5.9), which describes the dynamic behavior of the classes.

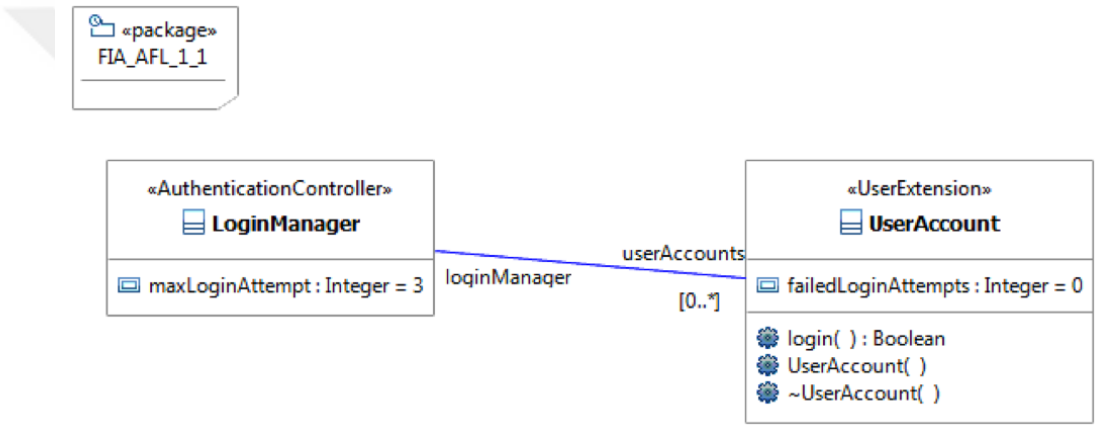


Figure 5.8: Model Checking UML Designs - Class Diagram

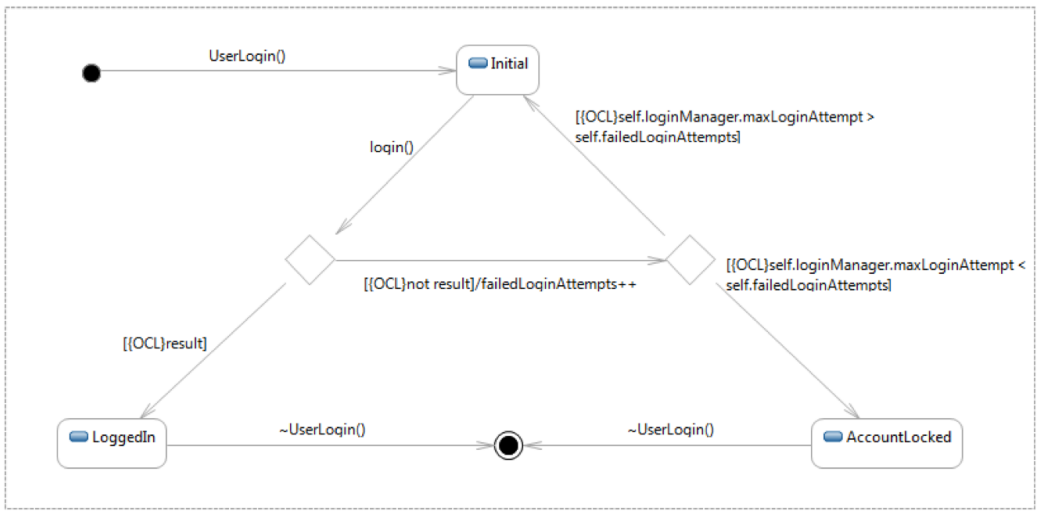


Figure 5.9: Model Checking UML Designs - State Machine Diagram

Details of above diagrams are explained in Section 4.3.2.2. However for this proof of concept, a small modification to the state machine is done by purpose.

5.4.4. Expected Outputs

Expected output of this proof of concept is a Boolean result which shows if design satisfies the requirement or not. An auxiliary output is the trace showing the steps of an execution, which causes the security property to be violated.

5.4.5. Execution Steps

5.4.5.1. Step 1 - Stereotyping the Design Model. In order to convert a UML model to PROMELA and execute model checking on the resulting model, input design model has to be annotated with necessary stereotypes.

These stereotypes are needed such that toolset can identify which attributes are behaviors are related to the security requirements. This will help model-checking tool to narrow down the state space of the execution.

Please refer to Section 4.3.2.2 for details of the applied stereotypes.

5.4.5.2. Step 2 - Executing PROMELA Generator Tool. PROMELA Generator Tool is a java application, which takes the design model's XMI output file's location as input. When the model in the proof of concept is input to the conversion tool a PROMELA model (Section B.2) is generated automatically.

This output PROMELA model is ready for model checking in the next step. If there were some problems, inconsistencies and wrong configurations in the input UML model, tool would have thrown exceptions stating the problem.

5.4.5.3. Step 3 - Writing LTL Property. The requirement explained in the proof of concept description, needs to be converted into a LTL property to be checked on the PROMELA model during verification. As described in Section 4.3.2.2, this security functional requirement can be converted to below LTL property:

[] ((failedLoginAttempts==3) -> (state==ACCOUNTLOCKED || state==FINAL))

Another LTL property to be checked on the PROMELA model would be to check value of failedLoginAttempts. By requirement this value shall not exceed maxLoginAttempt value: *[] ((failedLoginAttempts <= maxLoginAttempt)*

5.4.5.4. Step 4 - Executing Model Checking. When the first LTL property is executed on the SPIN model checker, below output is received:

State-vector 36 byte, depth reached 92, errors: 1
31 states, stored
2 states, matched
33 transitions (= stored+matched)
37 atomic steps

Execution of second LTL property is also resulted in error:

State-vector 36 byte, depth reached 109, errors: 1
36 states, stored
2 states, matched
38 transitions (= stored+matched)
43 atomic steps

5.4.5.5. Step 5 - Visualizing and Interpreting Model Checking Results. SPIN has a tool to automatically execute the resulting error trail using simulation capabilities.

When we look at the execution trail (Figure 5.10) and log of the execution (Table 5.4) it is evident that there's a problem with state transitions.

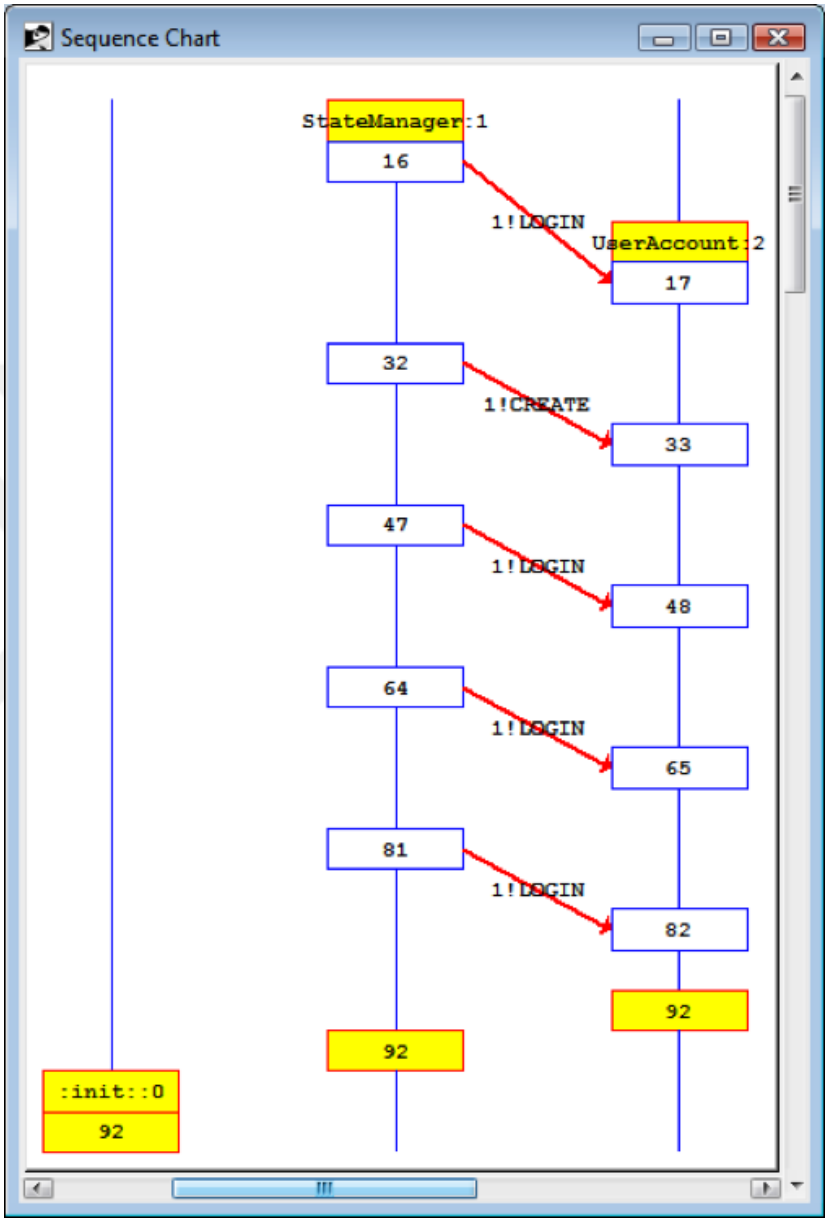


Figure 5.10: Execution Trail

Execution trail and logs shows that after 3 unsuccessful login attempts, state of the user is still INITIAL which creates a case where initial requirement is not satisfied.

It seems that guard condition in the state machine is not working properly such that whenever maximum number of unsuccessful login attempts is reached; state has

Table 5.4: SPIN Execution Log

Starting UserAccount with pid 3
CREATE event received 35: proc 2 (UserAccount) line 65 "pan_in" (state 33)
State is INITIAL 37: proc 2 (UserAccount) line 86 "pan_in" (state 57)
LOGIN event received 50: proc 2 (UserAccount) line 45 "pan_in" (state 10)
49: proc 2 (UserAccount) line 41 "pan_in" (state 4) [functionResult = 0]
State is INITIAL 54: proc 2 (UserAccount) line 86 "pan_in" (state 57)
LOGIN event received 67: proc 2 (UserAccount) line 45 "pan_in" (state 10)
66: proc 2 (UserAccount) line 41 "pan_in" (state 4) [functionResult = 0]
State is INITIAL 71: proc 2 (UserAccount) line 86 "pan_in" (state 57)
LOGIN event received 84: proc 2 (UserAccount) line 45 "pan_in" (state 10)
83: proc 2 (UserAccount) line 41 "pan_in" (state 4) [functionResult = 0]
State is INITIAL 89: proc 2 (UserAccount) line 86 "pan_in" (state 57)

to change to ACCOUNTLOCKED. However it does not stay at INITIAL state. Going back to the UML state machine diagram in the design model, it's easy to find that guard constraint for one of the transitions is not correct:

$self.loginManager.maxLoginAttempt < self.failedLoginAttempts$

This wrong guard condition causes the system to stay at INITIAL state even if max number of unsuccessful login attempts is reached. This error causes both LTL properties to fail. If this transition guard is corrected to below, model checking will not produce errors

$self.loginManager.maxLoginAttempt = self.failedLoginAttempts$

5.5. Proof of Concept 4: Model Checking Designs - Audit

5.5.1. Rationale

Aim of this proof of concept is to show how "MDSF Design Profile: Audit" is used for specification and verification of the Common Criteria Audit requirements.

5.5.2. Description

In this proof of concept, a software house is designing an SDK for audit log management functions and wants to verify if the SDK design model satisfies the Common Criteria audit security requirements:

- FAU_GEN.1.1: The TSF shall be able to generate an audit record of the following auditable events:
  - Start-up and shutdown of the audit functions;
  - All auditable events for the [selection, choose one of: minimum, basic, detailed, not specified] level of audit; and
  - Assignment: other specifically defined auditable events
- FAU_SAA.1.1: The TSF shall be able to apply a set of rules in monitoring the audited events and based upon these rules indicate a potential violation of the enforcement of the SFRs.
- FAU_ARP.1.1: The TSF shall take [assignment: list of actions] upon detection of a potential security violation.

5.5.3. Inputs

Inputs to this proof of concept are use case diagrams of the SDK and activity diagrams explaining the functionality of each use case.

Use case diagram in Figure 5.11 explains the functional capabilities of the SDK:

- ConfigureAudit: Function to set audit log configuration such as log level
- ExportAuditLogs: Function to select the logs and export them as a file
- VerifyLogsIntegrity: Function to check the integrity of the log records
- Log: Function to create an audit log
- SecurityAlarms: Function to analyse audit logs and raise an alarm if necessary
- Register: An external function with an auditable event

This use case diagram is annotated with "MDSF Design Profile: Audit" in order to show audit functions, auditable events, audit log function, and potential violation analysis use case and their relations to each other.

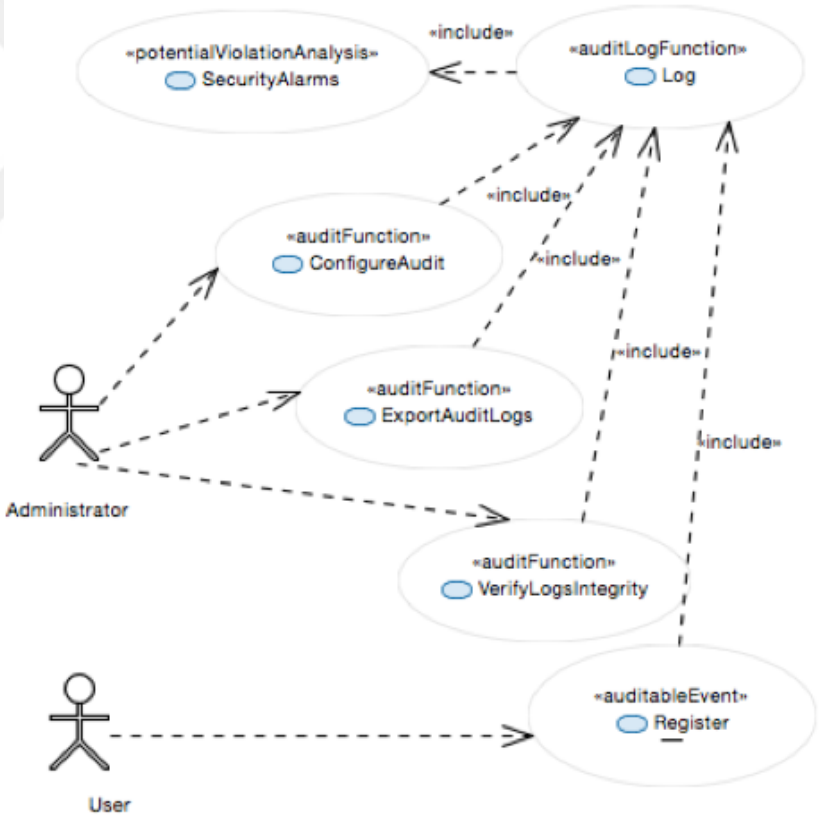


Figure 5.11: UML Use Case Diagram for Audit SDK

Activity Diagram in Figure 5.12 explains the high level activities of ConfigureAudit use case. This activity diagram is also annotated with "MDSF Design Profile: Audit" in order to show audit log steps during configuration.

Activity Diagram in Figure 5.13 explains the high level activities of Log use case. This activity diagram is also annotated with "MDSF Design Profile: Audit" in order to show log analysis steps during configuration.

Activity Diagram in Figure 5.14 explains the high level activities of SecurityAlarms use case. This activity diagram is also annotated with "MDSF Design Profile: Audit" and shows in what conditions a security alarm is generated.

5.5.4. Expected Outputs

Expected output of this proof of concept is a Boolean result which shows if design satisfies the requirements or not.

5.5.5. Execution Steps

FAU_GEN.1.1 can be verified on the input model by using the OCL invariants given in Table 4.22 and Table 4.23.

FAU_SAA.1.1 and FAU_ARP.1.1 are related to dynamic security properties and can only be verified by model checking. In Figure 5.14 a simple rule for potential violation analysis is given. The rule states that if number of invalid login events is 3, a potential violation event alarm is sent.

In order to verify if the rule is designed properly and raises an alarm when necessary, the activity model is converted to PROMELA using the PGT tool and model checked using Spin model checker. The following LTL property is verified on the PROMELA specification:

[] ((eventCount==3) -> (state==SendPotentialViolationEvent))

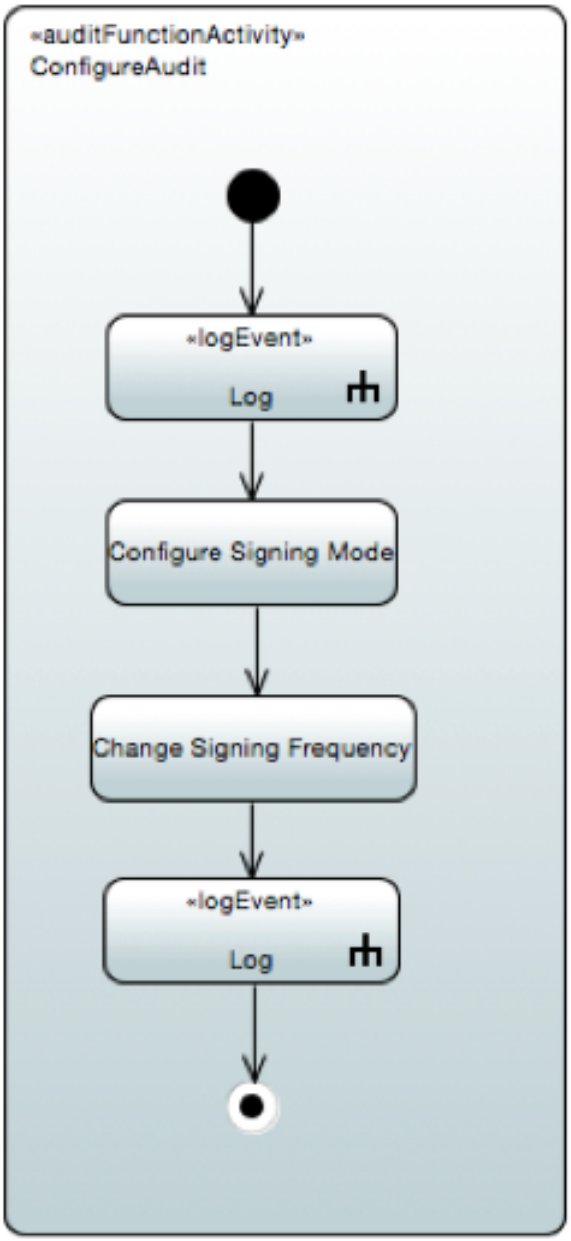


Figure 5.12: UML Activity Diagram for Audit Function

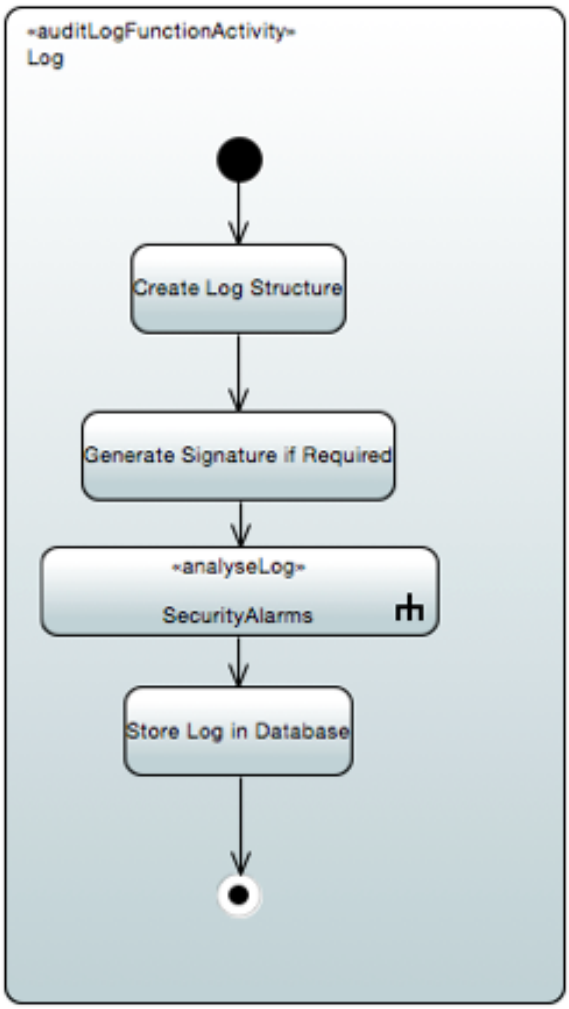


Figure 5.13: UML Activity Diagram for Audit Log Function

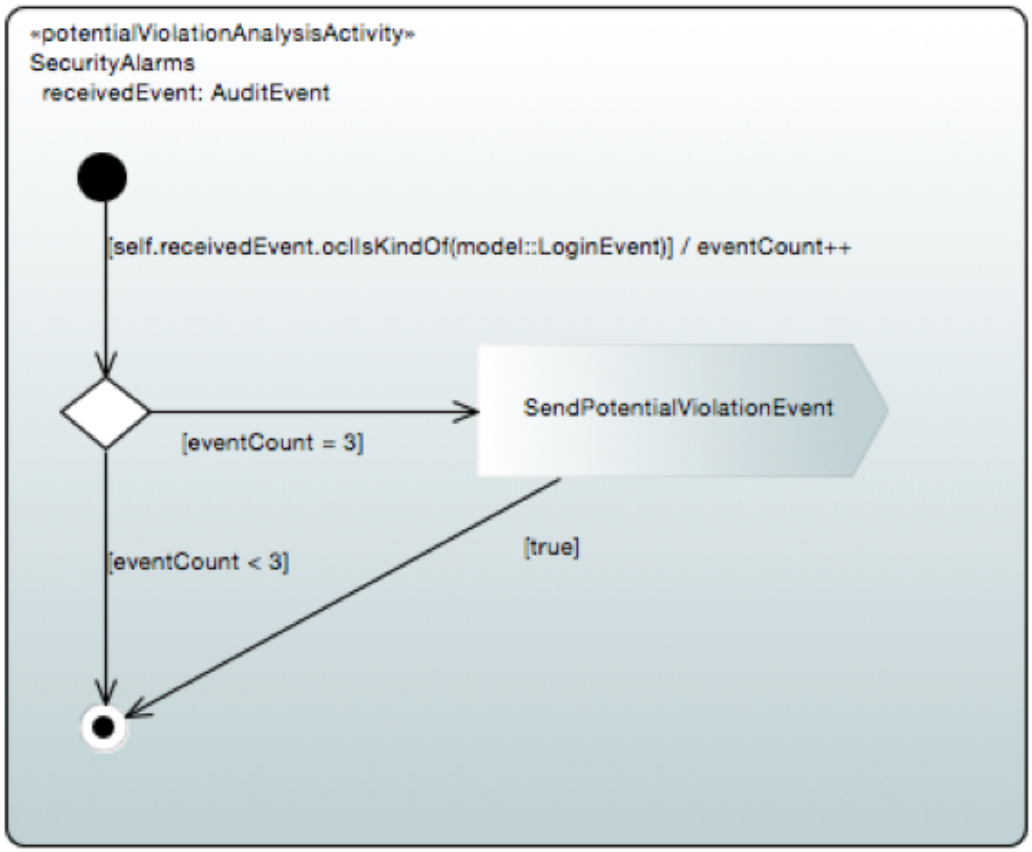


Figure 5.14: UML Activity Diagram for Potential Violation Analysis Function

5.6. Proof of Concept 5: Model Checking Designs - Information Flow

5.6.1. Rationale

Aim of this proof of concept is to show how "MDSF Design Profile: Information Flow" is used for specification and verification of the Common Criteria Information Flow requirements.

5.6.2. Description

In this proof of concept, a software house is designing a new mobile application to purchase, store and play music clips and wants to verify if their design satisfies the information flow security requirements of the Common Criteria.

5.6.3. Inputs

Inputs to this proof of concept are the class diagram and sequence diagram of the mobile application showing the information flow between the components of the mobile application and also between the mobile application and external service providers.

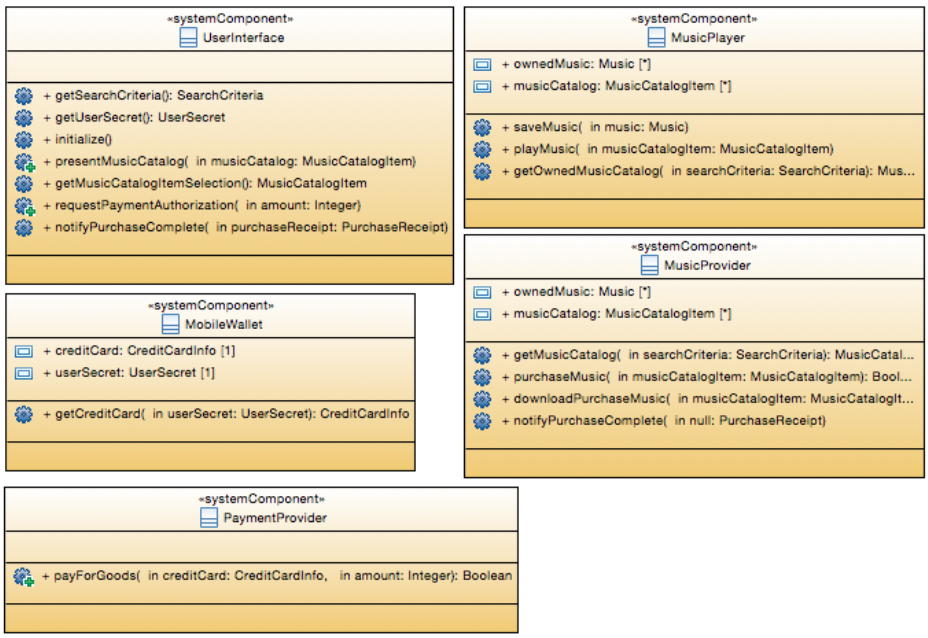


Figure 5.15: UML Class Diagram for Music Application

The class diagram in Figure 5.15 shows the components on the mobile device and also the external service providers. External service providers are MusicProvider and PaymentProvider. MusicProvider is responsible to deliver the music catalog and the music after the successfull payment. PaymentProvider is responsible to get the payment and notify the parties about the payment status.

The class diagram in Figure 5.16 shows the assets conveyed between the components on the mobile device as well as information conveyed to/from the external service providers.

Both of these class diagrams are annotated with "MDSF Design Profile: Information Flow" to set the TrustLevel of the components and the conveyed assets.

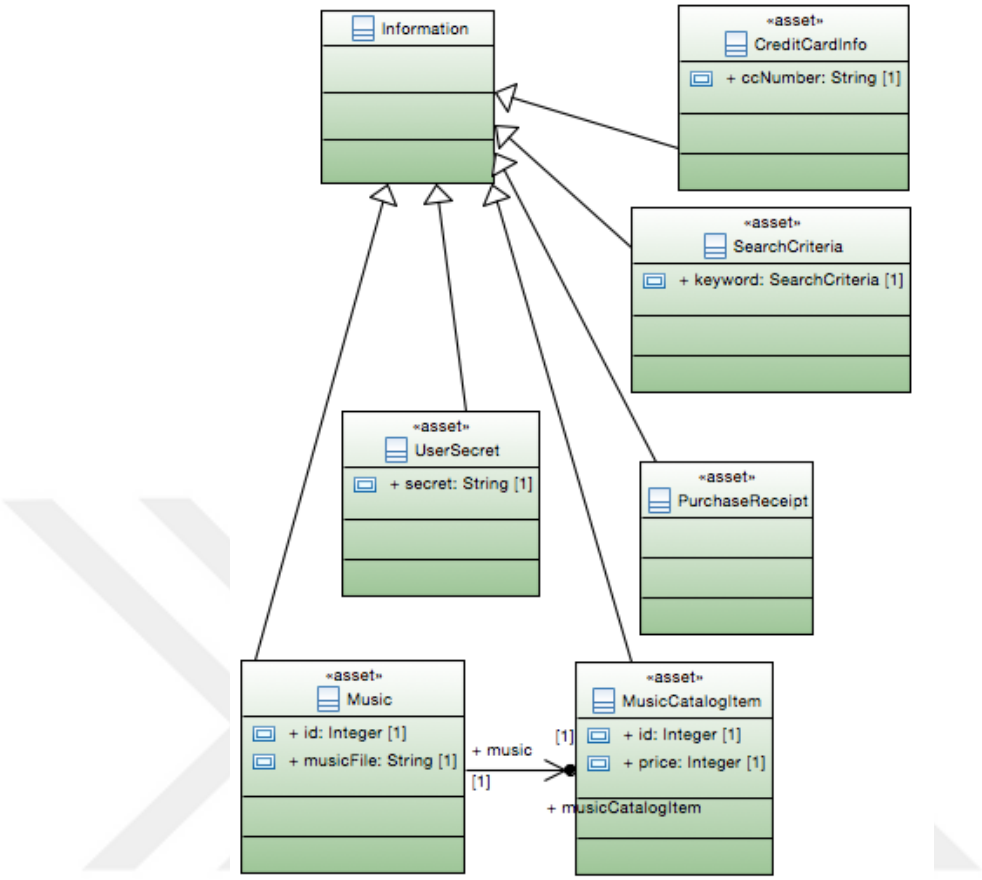


Figure 5.16: UML Class Diagram for Music Application

The sequence diagram in Figure 5.17 shows the information flow between different components. This sequence diagram is also annotated with "MDSF Design Profile: Information Flow" for the information flow requirements.

The following information flow requirements are to be enforced on this mobile application:

- FDP_IFF.1.1: The TSF shall enforce the [assignment: information flow control SFP] based on the following types of subject and information security attributes: [assignment: list of subjects and information controlled under the indicated SFP, and for each, the security attributes].
  - information flow control SFP 1: Trusted Information Flow
  - information flow control SFP 2: Fair Exchange Flow

Trusted Information Flow enforces a policy stating that the components can only receive information if the information's security level is equal or below the trust level of the component. According to this security policy, components cannot receive or store information with a higher security level.

Fair Exchange Flow enforces a policy stating that if the system is in state A, it should eventually go to state B. i.e. if the customer paid for the goods, he/she should eventually receive the goods in exchange.

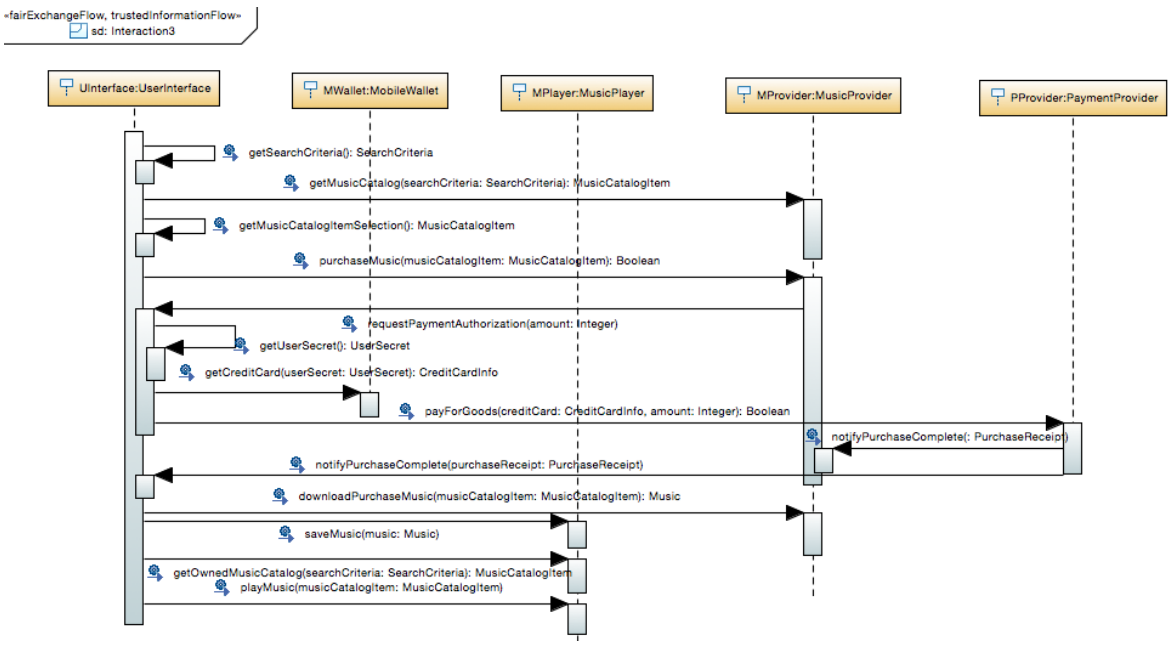


Figure 5.17: UML Sequence Diagram for Music Application

5.6.4. Expected Outputs

Expected output of this proof of concept is a Boolean result which shows if design satisfies the requirements or not.

5.6.5. Execution Steps

The sequence diagram is converted to a PROMELA specification (Section B.3) using the PGT tool.

The following LTL properties are verified using Spin in order to check Trusted Information Flow requirements:

failed: ltl check1 [] (mobileWalletReceived.trustLevel<=mobileWalletTrustLevel);

passed: ltl check2 [] (userInterfaceReceived.trustLevel<=userInterfaceTrustLevel);

passed: ltl check3 [] (musicPlayerReceived.trustLevel<=musicPlayerTrustLevel);

passed: ltl check4 [] (musicProviderReceived.trustLevel<=musicProviderTrustLevel);

passed: ltl check5 [] (paymentProviderReceived.trustLevel<=paymentProviderTrustLevel);

LTL property "check1" is failed since MobileWallet component has trust level 4 and UserSecret, which is conveyed to the MobileWallet has trust level 5.

The following LTL property is verified using Spin in order to check Fair Exchange Flow requirements:

passed: ltl fairexchange (<>(userInterfaceReceived.message==PurchaseReceipt) -> <> (musicPlayerReceived.message==Music)) && (musicPlayerReceived.message!=Music) U (userInterfaceReceived.message==PurchaseReceipt);

5.7. Proof of Concept 6: Model Checking Embedded Formal Descriptions

5.7.1. Rationale

Aim of this proof of concept is to show model transformation and model checking of embedded functional specification within class diagrams.

5.7.2. Description

In this proof of concept, a software house is designing a new web application, which will have a member only area that can be accessed via username and password. They want to get certified by ISO/IEC 15408 and use MDSF for that purpose. Designers created a class diagram for PasswordManager class which is responsible for generating user passwords and checking user generated passwords. Security analysts have identified one security functional requirement regarding the secrets:

FIA_SOS.2.1: The TSF shall provide a mechanism to generate secrets, which has length between 6 to 8 characters.

Security analysts want to make sure that the design really satisfies the defined security policy.

5.7.3. Inputs

Input to this proof of concept is a class diagram, which contains attributes and operations of a PasswordManager class (Figure 5.18).

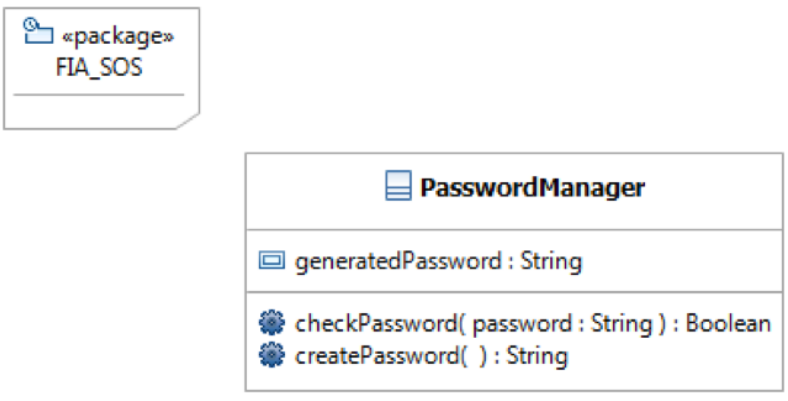


Figure 5.18: UML Class Diagram for Password Manager

Class diagram is annotated with MDSF UML profile stereotypes as shown in Figure 5.19. Operation which contains the formal representation of password generation

mechanism is stereotyped with “OperationWithFormalBody” stereotype.

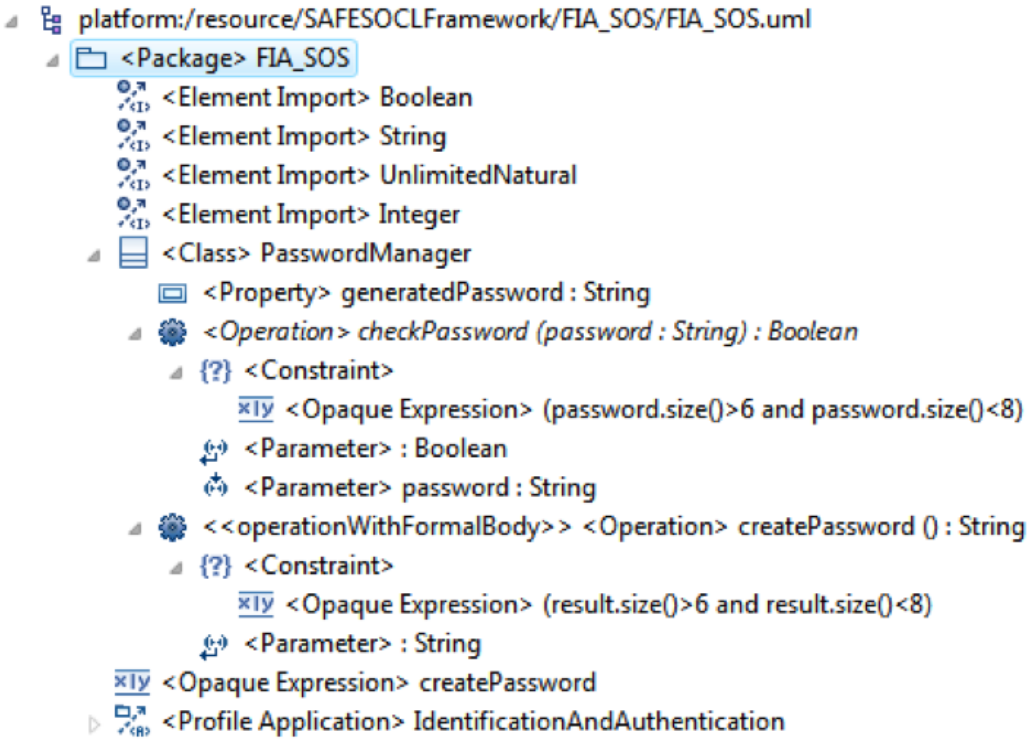


Figure 5.19: UML Class Diagram for PasswordManager - XMI format

createPassword operation contains “formalExpression” tagged value which contains an Opaque Expression written in C language (Figure 5.20).

5.7.4. Expected Outputs

Expected output of this proof of concept is a Boolean result which shows if design satisfies the requirement or not.

5.7.5. Execution Steps

5.7.5.1. Step 1 – PROMELA model generation. UML class diagram is input to the PROMELA Generator Tool as described in Section 4.4.2.3. Tool generates PROMELA model shown in Table B.4 as output.

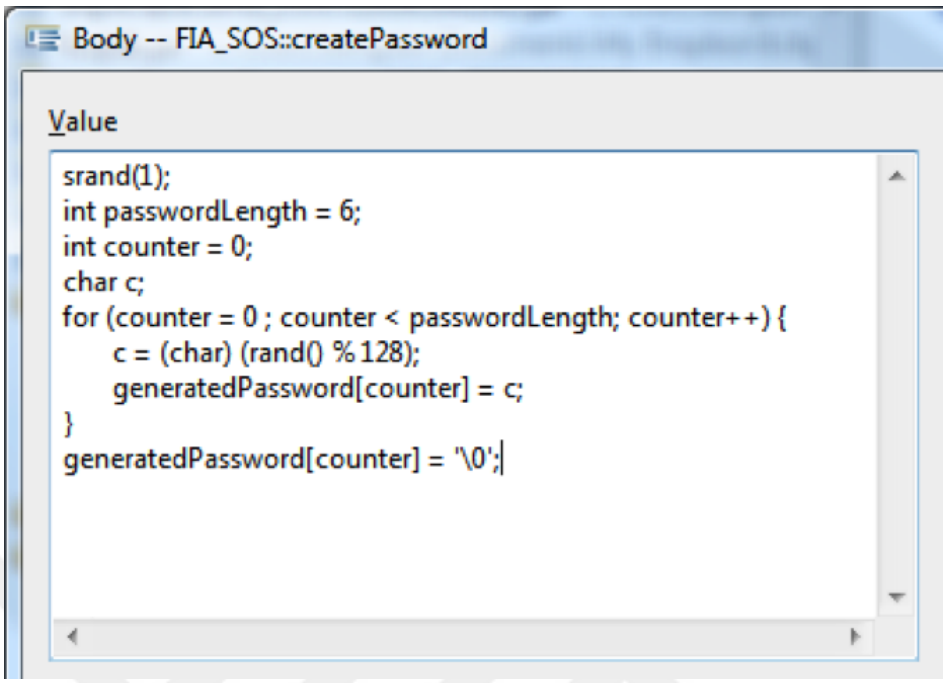


Figure 5.20: Formal Definition for createPassword Operation

5.7.5.2. Step 2 – OCL to LTL Conversion. Security functional requirement is written as a post condition on the operation within UML class diagram as described in Section 4.3.1.4:

context PasswordManager::createPassword(password: String): void
post: (result.size()>6 and result.size()<8)

Above OCL constraint is automatically converted to the corresponding LTL property below:

[] (p -> q)
#define p (createPasswordComplete)
#define q (generatedPassword[5]!=0 && generatedPassword[8]==0)

createPasswordComplete is an internal Boolean variable generated by the tool. This variable is set to true when the execution of the operation is completed.

Above LTL property states that whenever the password generation is completed, generated password length should be between 6 and 8.

5.7.5.3. Step 3 – Model Checking. As the last step of execution, both PROMELA model and LTL property are input to SPIN model checker. SPIN model checker shows if PROMELA model satisfies the LTL property or not.

State-vector 272 byte, depth reached 7, errors: 0
4 states, stored
1 states, matched
5 transitions (= stored+matched)
0 atomic steps
hash conflicts: 0 (resolved)

5.8. Proof of Concept 7: Mobile Application Store for a Telecom Operator

In this proof of concept, we show how a fictious software development house can use MDSF during the International Standards Organization (ISO) 15408 Common Criteria certification process (EAL6) for their new application store software product.

The application store allows subscribers of a service provider to access the mobile application repository through different channels such as mobile internet, web and mobile client. Subscribers can surf the application store, purchase or gift applications using their prepaid/postpaid accounts and download the purchased applications to their mobile phones. Apart from the subscribers, other stakeholders of this application store include administrators and partners. Administrators are responsible for managing the content lifecycle of the application store and obtaining statistical and commercial reports. The UML use case diagram for the application store is given in Figure 5.21.

During the proof of concept, we will show usage of MDSF (Figure 4.15) through seven steps starting from creation of an analysis model until the formal verification of the security requirements.

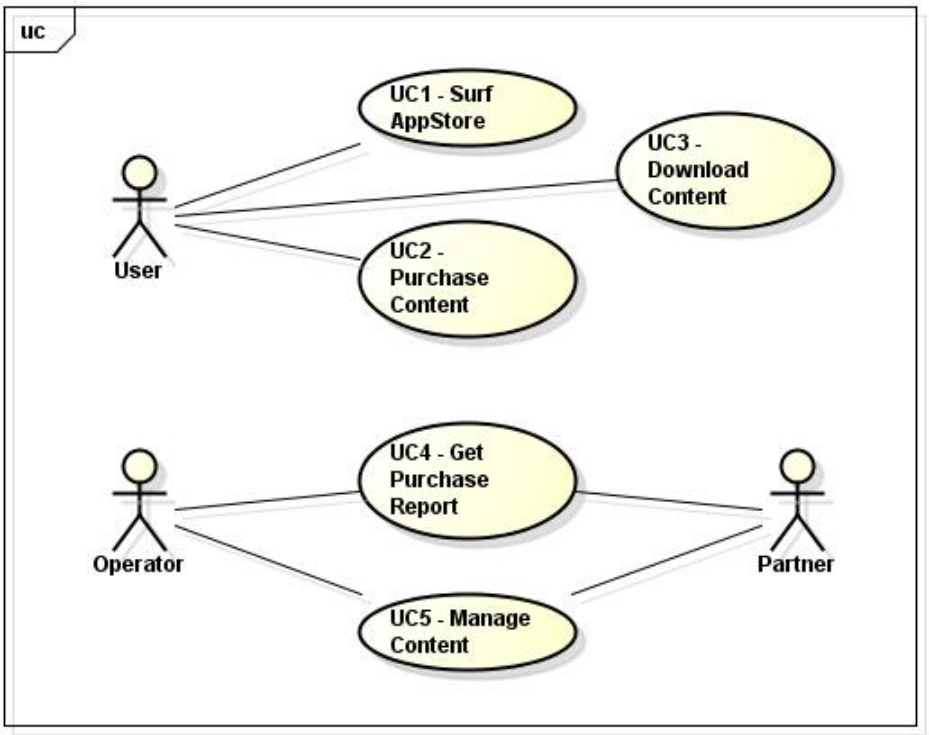


Figure 5.21: Use Case Diagram for App Store

5.8.1. Step 1: Creation of Analysis Model

The AcmeSWR analyst has generated four necessary diagrams for the first step. These diagrams are Use Case Diagram (Figure 5.21), Critical Asset Diagram (Figure 5.22), Communication Diagram (Figure 5.23) and Use Case Description Diagram (Figure 5.24).

The Critical Asset Diagram (Figure 5.22) shows the information assets associated with the PrepaidUser actor and classifies them based on the impact of a security breach. Credentials and CommerceEvent assets have high impact, whereas Session and PersonalInformation assets have medium impact. The MSISDN asset is marked as low impact because the disclosure of MSISDN does not cause much trouble. The MSISDN asset is also marked as a unique identifier for the PrepaidUser actor.

The Communication Diagram (Figure 5.23) shows how different modules of the system are connected with each other in terms of network connections. For the mobile application store, the User actor is connected to the AppStore module and the CMS module via an untrusted communication line. Other modules are in the local network

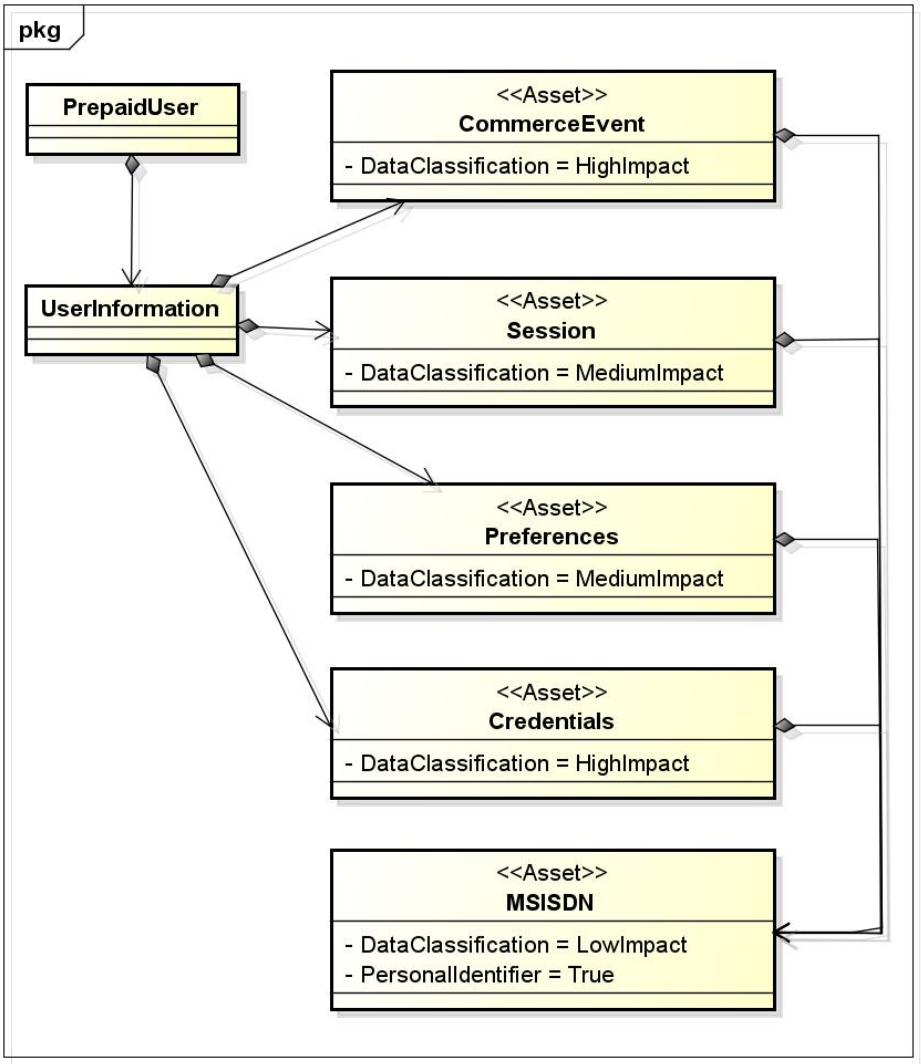


Figure 5.22: Critical Asset Diagram for App Store

of the operator and hence communicate via a trusted line.

The Use Case Description Diagram (Figure 5.24) shows which critical assets are transferred between different modules during steps of a use case. The session asset is transferred from PrepaidUser actor to AppStore module, and the CommerceEvent asset is transferred from Appstore to ChargingSystem module and Partner actor.

5.8.2. Step 2: Identification of Threats with TIT

In Step 2, the AcmeSWR analyst inputs Use Case Diagram (Figure 5.21), Critical Asset Diagram (Figure 5.22), Communication Diagram (Figure 5.23) and Use Case

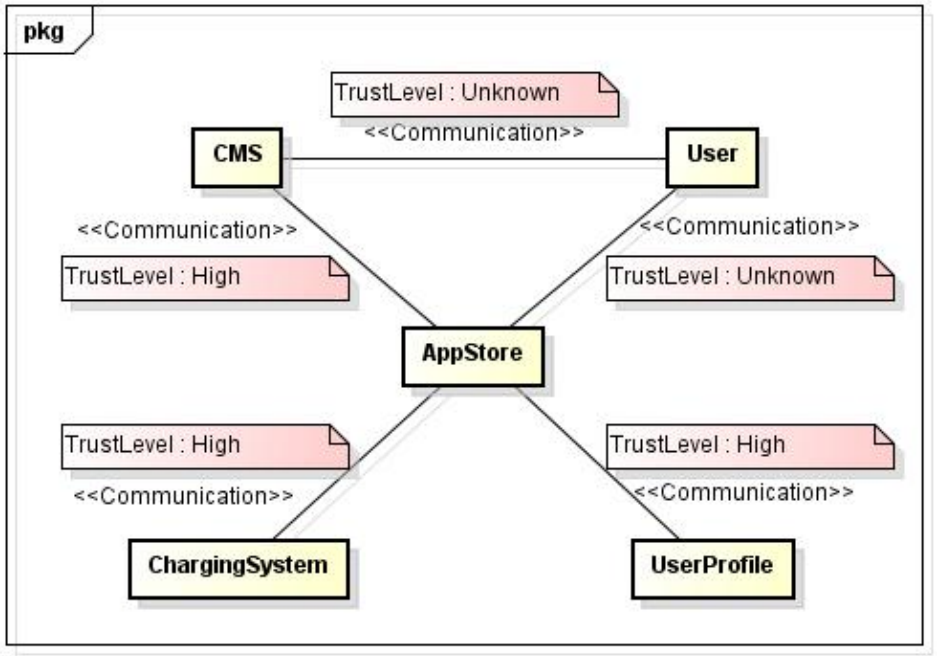


Figure 5.23: Communication Diagram for App Store

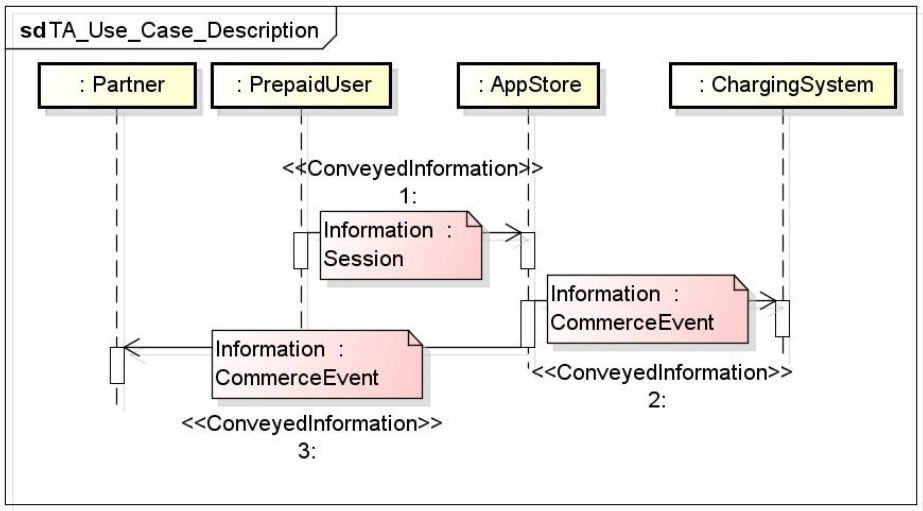


Figure 5.24: Use Case Description Diagram for App Store

Description Diagram (Figure 5.24) into the TIT of MDSF. The TIT executes stored OCL statements on the input models and identifies the following threats:

- T.Outsider: An individual who is not an authorized user of the system may gain access to the TOE. This threat is selected because assets (e.g., Session, Credentials) with medium or high impact classification are transferred on an untrusted communication line.

- T.Privacy_Violated: Unauthorized access to privacy data of system users may occur without detection. This threat is selected because assets (CommerceEvent) containing a personal identifier (MSISDN) belonging to one actor are transferred to another actor. This type of information flow may result in privacy violations if not controlled.
- T.Logon_Attack: A program that tries a large number of passwords to guess the password of an authorized user. The Attack allows an unauthorized individual to log on as an authorized user and assume the identity of that user. This threat is selected because a T.Outsider threat is identified. T.Outsider requires authentication to be implemented. Authentication of an actor generally starts with a logon function, which is subject to logon attacks.

5.8.3. Step 3: Extraction of Security Requirements with SRET

The AcmeSWR analyst inputs the threats identified in Step 1 to SRET of MDSF. SRET matches the threats to the security objectives listed in Table 5.5 using the MDSF Repository.

Each security objective in Table 5.5 is also decomposed into SFRs by SRET. The list of identified SFRs is given in Table 5.6.

5.8.4. Step 4: Creation of Design Model

In Step 4, the AcmeSWR analyst hands over the identified SFRs to the designer. The designer creates a detailed class diagram for the App Store (Figure 5.25, Figure 5.26). The designer considers the identified SFRs and creates a design for not only the functional requirements but also the security requirements. One of the SFRs identified is FDP_ACC.1.1. FDP_. ACC.1.1 requires TSF to enforce access control for selected operations. The designer annotates the class diagram using the MDSF Design Profile to define the access control model, identify roles and access rights, annotate the operations that require access control restrictions and mark conflicting access rights.

Table 5.5: Security Objectives for Identified Threats

Security Objective	**Description**	**Threat**
O.Authen	Users must be uniquely identified and authenticated prior to performing any actions to be mediated by the TOE	T.Outsider, T.Privacy_Violated, T.Data_Theft
O.Access Control	The TOE must enforce an access control policy to protect information from unauthorized access	T.Outsider, T.Privacy_Violated, T.Data_Theft
O.Anon	The TOE must provide for anonymity and pseudonymity	T.Privacy_Violated
O.Logon Limit	The TOE must limit the number of logon attempts	T.Logon_Attack

The designer creates a multilevel state diagram, sequence and state machine diagrams for the major use cases of the system (Figures 5.27, 5.28, 5.29, and 5.30) to design the functional behavior.

The multilevel state machine diagram (Figure 5.27) shows the main events exchanged between different components of the system.

The sequence Diagram for the App Store (Figure 5.28) shows the interaction of different system components for surfing, purchasing and downloading use cases. The designer also defines a simple authentication flow for the prepaid user in this diagram.

The Sequence Diagram for Content Management Operations (Figure 5.29) shows the interactions of different system components for content lifecycle operations such as ingestion, modification and approval.

The State Machine Diagram for Prepaid User (Figure 5.30) defines the states

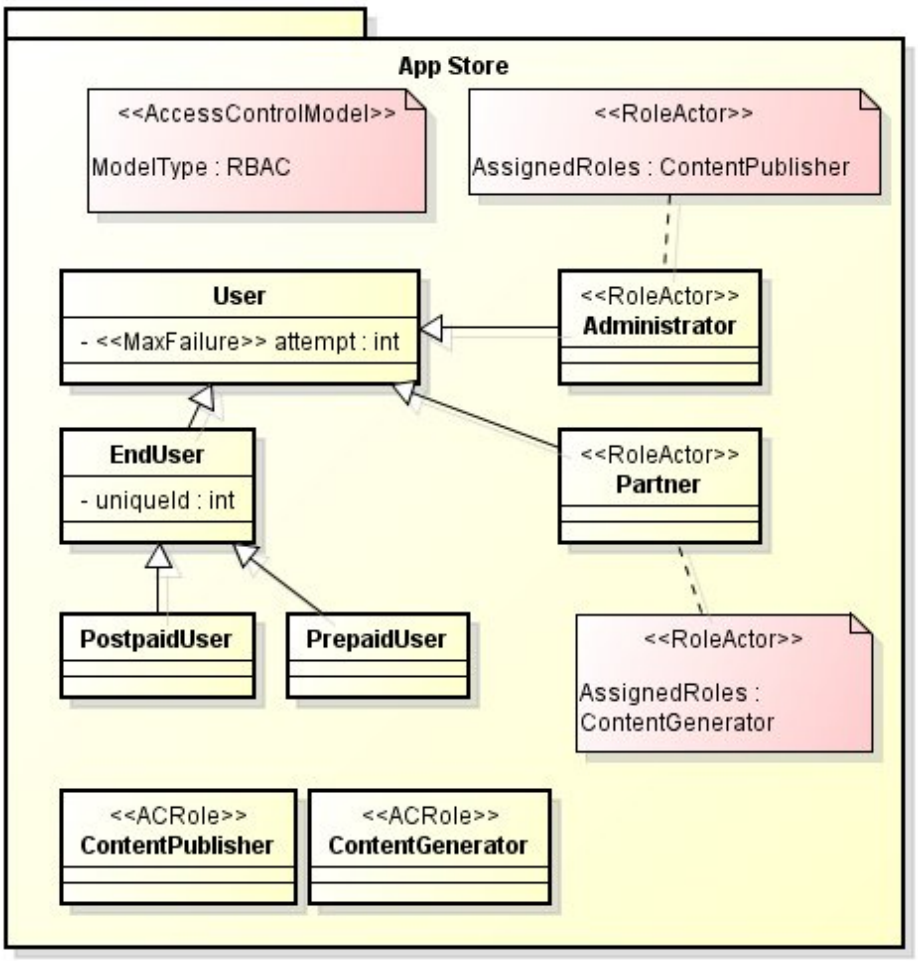


Figure 5.25: Class Diagram for App Store - Users

of prepaid user actor and changes in states for different triggers. Prepaid User actor starts in the "Idle" state. The state changes to "PublicSurfing" when the actor starts surfing the App Store. This state machine shows all the possible states and all the triggers that cause the state changes for the Prepaid User actor. These triggers can be internal triggers generated by Prepaid User actions or external triggers generated by events coming from other system components. AcmeSWR designer also annotates this diagram for FIA_AFL.1.1 and FIA_AFL.1.2 SFRs using the MDSF UML Design Profile.

5.8.5. Step 5: Enhancement of Model with SFMT

In this step, the AcmeSWR designer uses SFMT of MDSF to enhance the design of the system with pre-verified security function designs from the MDSF Repos-

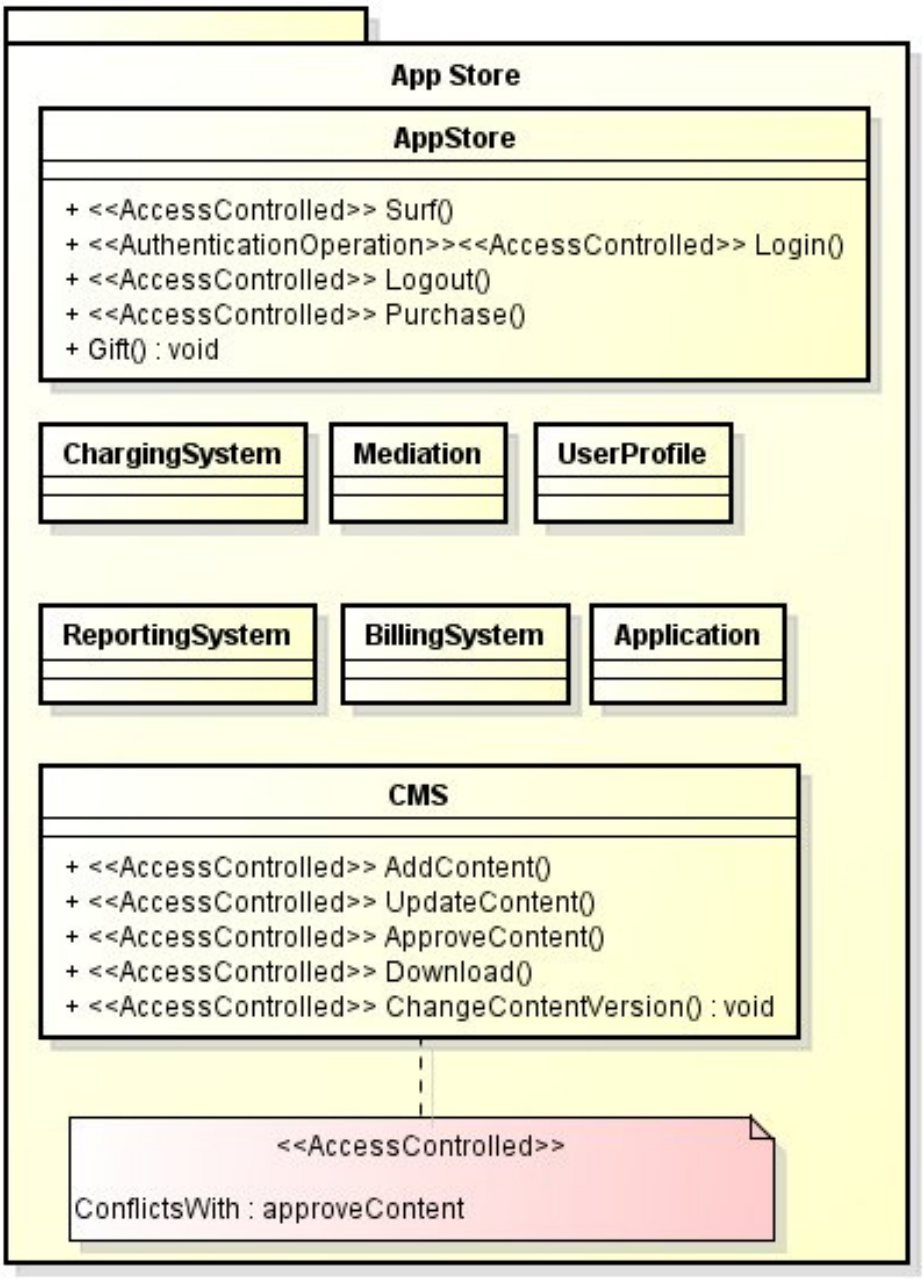


Figure 5.26: Class Diagram for App Store - Modules

itory. Figure 5.31 shows a pre-verified security function design for FIA_AFL.1.1 and FIA_AFL.1.2 of the Common Criteria. This state machine limits the number of unsuccessful login attempts for the Prepaid User actor. If the number of unsuccessful login attempts exceeds a predefined number, Prepaid User state is changed to "AuthenticationFailure" state. Designer will reduce the time spent for design activity by reusing this state machine and merging it to the main design model of the information system. When the designer inputs the design model to SFMT, merge points are analyzed with

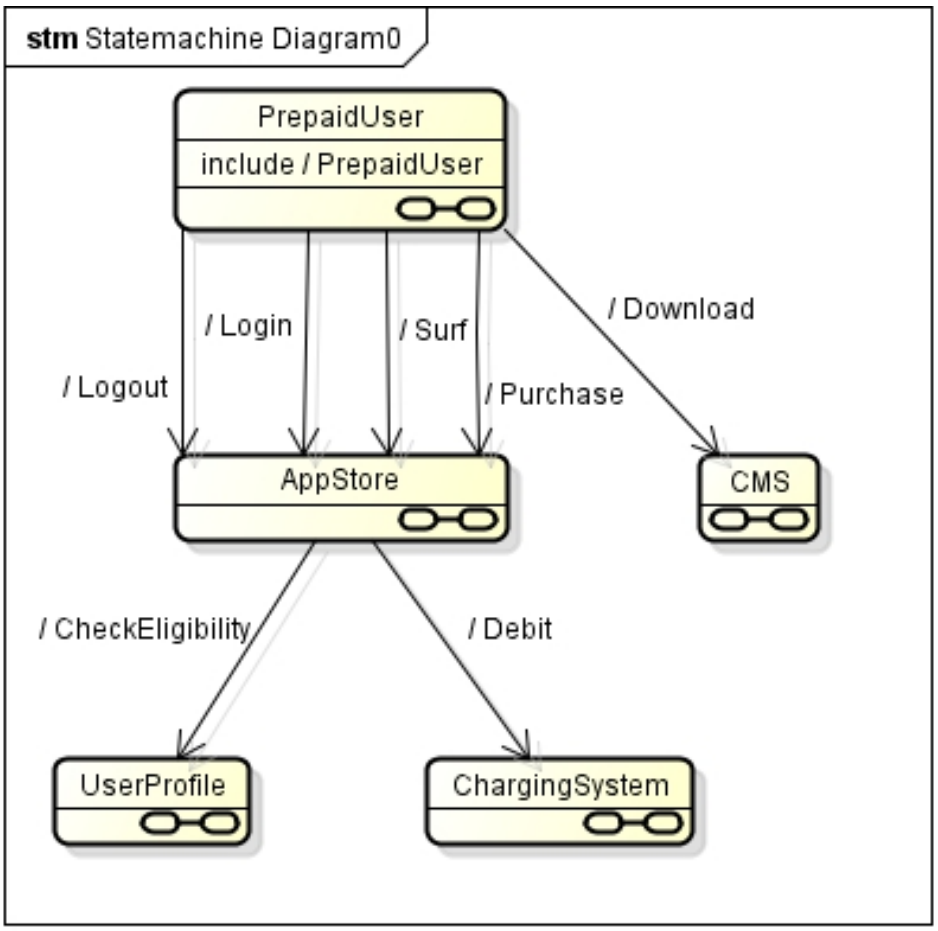


Figure 5.27: Multi-Level State Machine Diagram for App Store

the help of the associated stereotypes and tagged values on the design model. For the Prepaid User State Machine in Figure 5.30, merge points are defined by the AuthenticationFailureHandling stereotype. Necessary parameters for the merge operation are provided by the tagged values of the AuthenticationFailureHandling stereotype.

After identifying the merge points and the security function design to merge, SFMT executes the merge operation and generates a new design diagram, as shown in Figure 5.32. This diagram includes the Prepaid User actor states defined in the state machines of Figure 5.30 and Figure 5.31. SFMT connects these state machines to each other by taking the triggers and merge points into consideration. At this step, the AcmeSWR designer can continue to enhance the design model with additional stereotypes using the MDSF UML Design Profile. The designer adds the TimingOfAuthentication stereotype and necessary tagged values, as shown in Figure 5.32. The TimingOfAuthentication stereotype helps the designer annotate the design model for

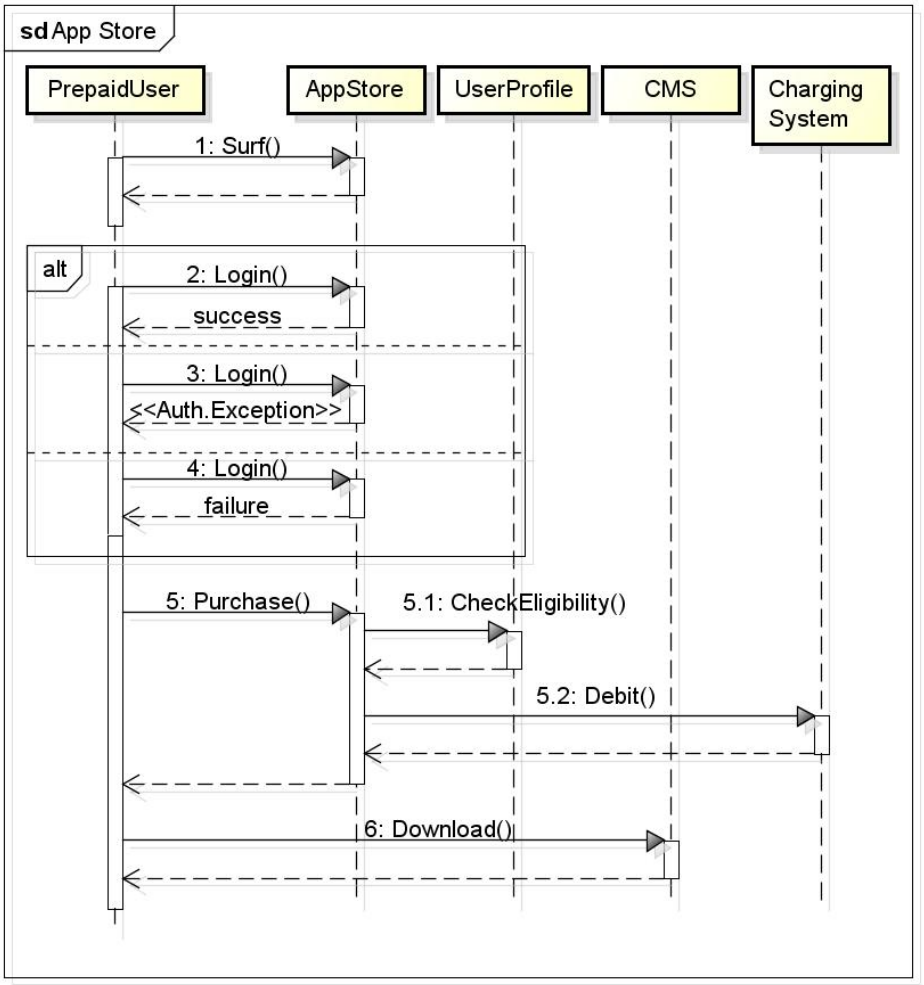


Figure 5.28: Sequence Diagram for App Store

FIA_UAU.1.1 and FIA_UAU.1.2 requirements from Table 5.6.

5.8.6. Step 6: Generation of PROMELA Model

At the end of the design phase, the AcmeSWR designer or tester uses the PGT of MDSF to convert the complete design model to a PROMELA specification with embedded LTL properties for corresponding security requirements. The PROMELA specification is required as input in the next step to model-check the design model and thus satisfy security requirements.

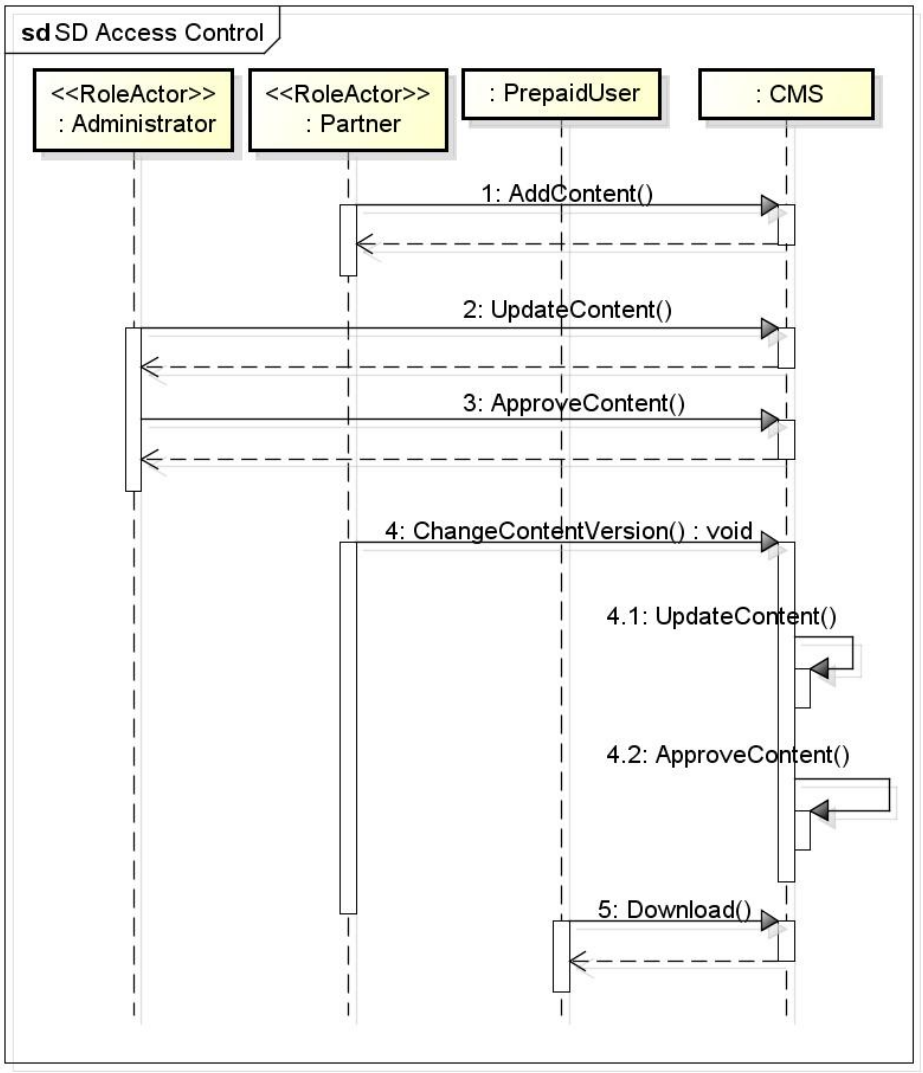


Figure 5.29: Sequence Diagram for Content Management Operations

5.8.7. Step 7: Static and Dynamic Model Checking

In Step 7, the AcmeSWR designer or tester inputs the PROMELA specification into the Spin Model Checker. The Spin Model Checker executes model-checking algorithms and generates counterexamples for the LTL properties that are not satisfied. The AcmeSWR designer or tester analyzes the counterexamples using the simulation capability of the Spin Model Checker and reaches the following conclusions:

- FIA_AFL.1.1 and FIA_AFL.1.2: There is no counterexample generated for these requirements, meaning that the security function design is merged correctly in Step 5 and satisfies the requirements.

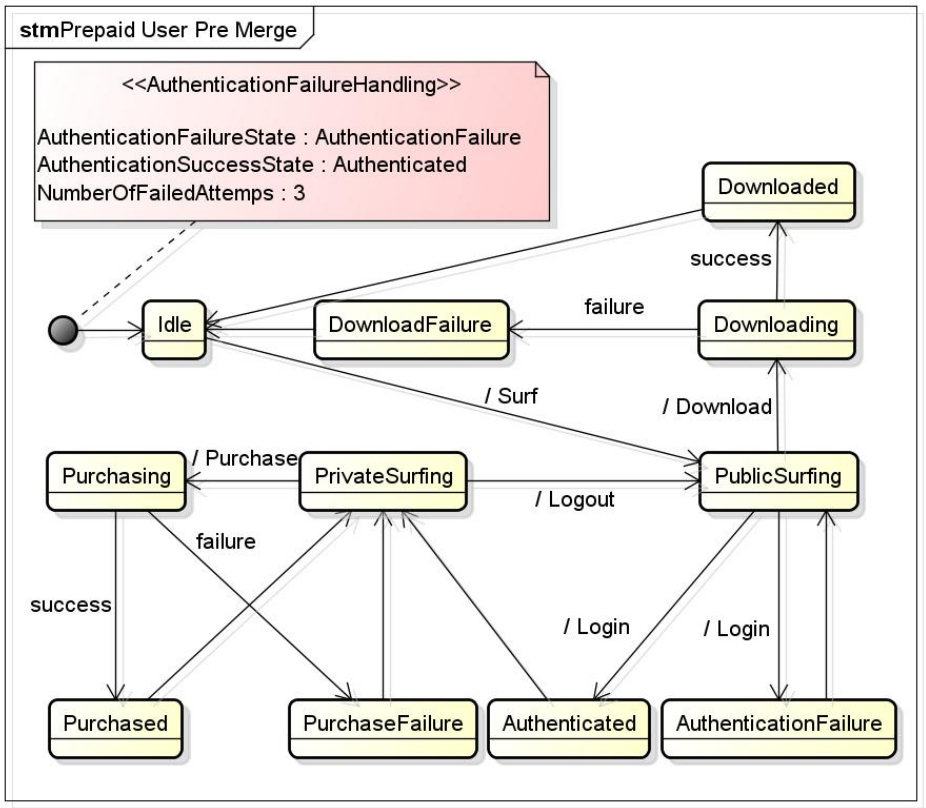


Figure 5.30: State Machine Diagram for Prepaid User

- FIA_UAU.1.1 and FIA_UAU.1.2: The TimingOfAuthentication stereotype puts the requirements on the state machine diagram to control which states are allowed before and after the successful authentication of the user (Figure 5.32). However, the Spin Model Checker identifies a counterexample showing that the Downloaded, Downloading, Downloaded states can be reached before a successful authentication in the State Machine Diagram for the Prepaid User. This counterexample conflicts with the TimingOfAuthentication stereotype and hence does not satisfy the FIA_UAU.1.1 and FIA_UAU.1.2 requirements.
- FDP_ACC.1.1: In the Class Diagram of Figure 5.26, the ChangeContentVersion operation of the CMS class is marked with the AccessControlled stereotype, and the ConflictsWith tag has the approveContent operation as the value. This annotation means that an actor that has the access right for ChangeContentVersion operation cannot have the access right for the approveContent operation. However, the Spin Model Checker creates a counterexample showing that the Diagram for Content Operations (Figure 5.29) contains an operation chain where the Part-

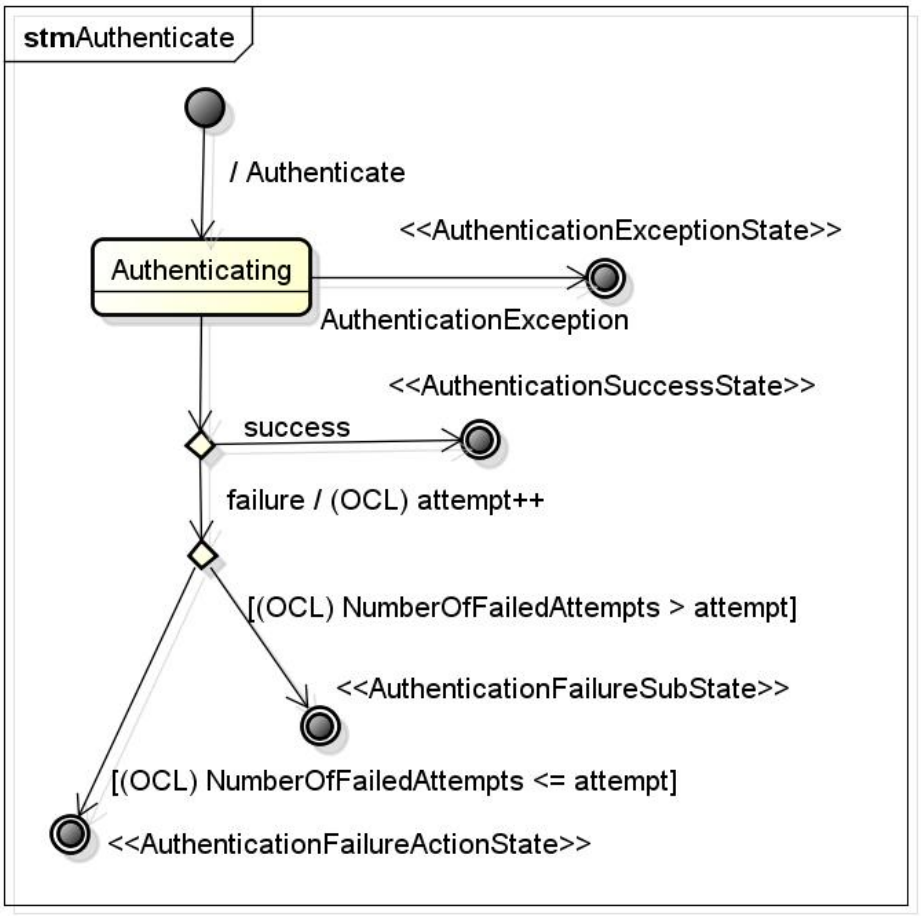


Figure 5.31: Security Function Design for Authentication

ner actor executes both ChangeContentVersion and ApproveContent operations in sequence despite the fact that these two are conflicting operations as stated by the access control policy.

At the end of Step 7, the AcmeSWR team analyzes the above counter examples and enhances the analysis model and design model accordingly by executing Steps 1 to 7 until the design model satisfies all of the security requirements.

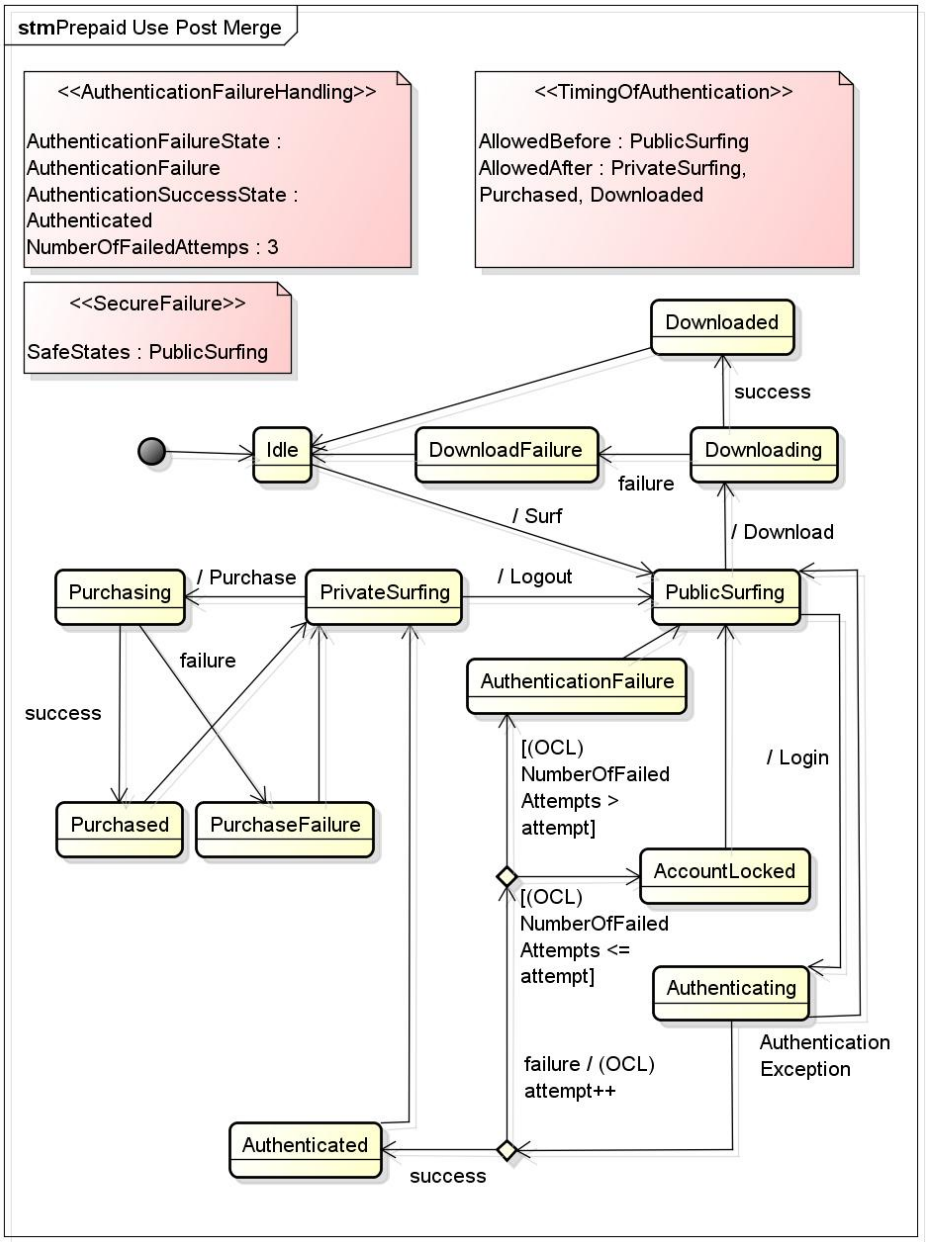


Figure 5.32: State Machine Diagram for Prepaid User after SFM

Table 5.6: SFRs extracted from Security Objectives

Req.	**Description**	**Objective**
FIA_AFL .1.1	The TSF shall detect when [selection: [assignment: positive integer number], an administrator configurable positive integer within [assignment: range of acceptable values]] unsuccessful authentication attempts occur related to [assignment: list of authentication events]	O.Logon
FIA_AFL .1.2	When the defined number of unsuccessful authentication attempts has been [selection: met, surpassed], the TSF shall [assignment: list of actions]	O.Logon
FIA_UAU .1.1	The TSF shall allow [assignment: list of TSF mediated actions] on behalf of the user to be performed before the user is authenticated	O.Authen
FIA_UAU .1.2	The TSF shall require each user to be successfully authenticated before allowing any other TSF-mediated actions on behalf of that user	O.Authen
FPR_ANO .1.1	The TSF shall ensure that [assignment: set of users and/or subjects] are unable to determine the real user name bound to [assignment: list of subjects and/or operations and/or objects]	O.Anon
FDP_ACC .1.1	The TSF shall enforce the [assignment: access control SFP] on [assignment: list of subjects, objects, and operations among subjects and objects covered by the SFP]	O.Access _Control

6. CONCLUSIONS

In this thesis we propose a framework called Model Driven Security Framework (MDSF) utilizing UML, OCL, PROMELA and Spin and show how we can support the higher-level (Levels 6 and 7) assurance processes of the Common Criteria by using these languages and formal methods on the analysis and design models of an information system.

In summary, major contributions of MDSF are as follows:

- A framework with necessary tools to support the analysis and design of security features of an IT product and ease the Common Criteria certification process with formal verification of security policies on the design model.
- A tool and a new UML profile (MDSF Threat Analysis Profile) supported with OCL invariants for automated threat analysis and identification
- A tool for extraction of security objectives and security functional requirements from the identified security threats
- New UML profiles to design for identification and authentication, access control, audit and information flow security requirements in the Common Criteria
- A tool for merging already available security function designs into a design model
- OCL queries and invariants to check the design model for completeness, conformance and security requirements
- A tool for transforming the UML design models into PROMELA specifications to be checked in the Spin model checker.

The application of MDSF to analysis and design of software systems is also presented in this thesis. Usage of all MDSF tools and methods are explained on several hypothetical proof of concepts:

- Proof of Concept 1: Explains the MDSF Threat Analysis Profile and Threat Identification Tool and their application on an example model.

- Proof of Concept 2: Explains the security function repository and Security Function Merger Tool and their application on an example model.
- Proof of Concept 3: Explains the MDSF Identification and Authentication Profile, PROMELA Generator Tool and verification of security requirements via model checking.
- Proof of Concept 4: Explains the MDSF Audit Profile, OCL invariants for static requirement verification, PROMELA Generator Tool and verification of security requirements via model checking.
- Proof of Concept 5: Explains the MDSF Information Flow Profile, PROMELA Generator Tool and verification of security requirements via model checking.
- Proof of Concept 6: Explains the embedded formal specifications in the design model and how they are converted to PROMELA and verified using model checking
- Proof of Concept 7: Explains all of the MDSF tools and processes in action

There are several possible improvement areas for the MDSF. In the center of the framework, we used a new UML Profile to enable security function analysis and design. The MDSF UML Profile is not complete and currently supports a subset of security classes of the Common Criteria. The following security classes can be added in the future work:

- Communication
- Cryptographic Support
- Security Management
- Privacy
- Protection of the TSF
- Resource Utilization
- TOE Access

PROMELA Generator Tool currently supports a subset of UML diagram types including state machines, sequence diagrams and class diagrams. Other diagram types

and UML 2.0 model elements can be added in the future to improve the PROMELA Generator Tool capability. PROMELA Generator Tool transforms the UML diagrams into PROMELA specifications. In the current version of the framework, correctness of this transformation is not formally verified. As a future work, formal verification of the soundness of the model transformation can help MDSF to be better positioned for the security certification evaluation process. The process of model transformation is automated in MDSF. However, there's no tool proposed in MDSF to interpret the model checking results. Spin model checker generates an error trail which can be visualized in xspin or ispin tool. Automatic generation of a UML diagram showing the error state of the software model can help the designers to better understand why the security requirement is not satisfied.

Threat Anaysis Tool uses the OCL invariants in the MDSF Threat Library in order to identify possible threats for the software system. For each possible threat there exists at least one OCL invariant in the MDSF Threat Library. However in the current version of the framework, only a limited number of threats are defined. As a future work, OCL invariants for all the threats listed in Table A.1 can be specified in order to improve the threat identification process.

APPENDIX A: THREATS AND SECURITY OBJECTIVES

Table A.1: Threat Definitions

Threat Name	**Threat Description**
T.Actions_Traced	Unauthorized tracing of customer business actions may occur without detection.
T.Blockage	Two systems may not be able to exchange data due to a communications channel being blocked.
T.Change_Data	Information may be changed either while it being stored or processed within the TOE or during transmission. The changes may be accidental or intentional. Changes include insertions, replacements, modifications, and deletions. The type of information that can be changed includes user information, system information, business data, and commitment.
T.Comm_Failure	It may not be possible to set up a connection or transmit data between two systems.
T.Data_Theft	Business process data may be stolen.
T.Deny_Service	Application and network services may not be available for use.
T.Deny_Data	An entity may deny ownership of business or commitment data.
T.Deny_Receipt	An entity may deny that it has received business or commitment data.

Continued on next page

Table A.1 – continued from previous page

Threat Name	Threat Description
T.Deny_Submit	An entity may deny that it has submitted business or commitment data.
T.Disaster	Natural disasters may cause TOE or system failure.
T.Disclose_Data	Information may be disclosed in to unauthorized users. This includes both user information, system information and business data. Information may be disclosed while being stored or processed in the TOE or during transmission. Disclosure of authentication data during transmission would allow someone to logon and assume the identity of an authorized user.
T.False_Routing	Information may be routed to a false address enabling unauthorized access.
T.History_Untraceable	It may not be possible to trace the sequence of events during a system failure, malfunction, or betrayal.
T.Impersonate	Someone may obtain unauthorized access by impersonating an unauthorized user, for example by hijacking a communications session.
T.Indeterminate_Seq	It may be impossible to determine the sequence of events in a dispute due to missing time information and business related data.

Continued on next page

Table A.1 – continued from previous page

Threat Name	Threat Description
T.Insider	An authorized user of the TOE may gain unauthorized access.
T.Intercept	Commitment data or certificates may be intercepted.
T.Invalid_Certificate	The TOE may accept invalid commitment data or certificates.
T.Logon_Attack	A program that tries a large number of passwords successfully guesses the password of an authorized user. This allows an unauthorized individual to logon as an authorized user and to assume the identity of that user.
T.Outsider	An individual who is not an authorized user of the system may gain access to the TOE.
T.Physical	A component of the system may be accidentally or intentionally damaged.
T.Privacy_Violated	Unauthorized access to privacy data of system users may occur.
T.Refusal	Unauthorized refusal of valid commitment data or certificates may occur.
T.Replace_TOE	The TOE may be replaced by an untrusted system.
T.Replay	Someone may obtain unauthorized access by replaying authentication or commitment data.

Continued on next page

Table A.1 – continued from previous page

Threat Name	Threat Description
T.Secret_Disclose	Authentication information may be disclosed allowing someone to logon and assume the identity of an authorized user.
T.Service_Denied	Application and network services may not be available for use.
T.TOE_Fail	TOE or system failure may cause the TOE to enter an insecure state or data to be disclosed or changed.
T.Traffic	A system may experience degraded performance due to increased communications traffic.
T.Unique_Copied	Unlawful copies may be made of unique originals.

Table A.2: Security Objectives

Objective Name	Objective Description
O.Access_Control	The TOE must enforce an access control policy protects information from unauthorized access.
O.Alt_Channel	The TOE must provide alternate channels for the transmission of data.
O.Anon	The TOE must provide for anonymity and pseudonymity.

Continued on next page

Table A.2 – continued from previous page

Objective Name	Objective Description
O.Assoc_User_Action	The TOE shall associate all security relevant actions with a unique user identity.
O.Attest	The TOE must provide for attestation of submission, delivery, and receipt
O.Audit	All security relevant actions must be auditable.
O.Authen	Users will be uniquely identified and authenticated prior to performing any actions to be mediated by the TOE.
O.Authen_Address	The TOE must be able to authenticate sender and receiver addresses. This includes a priori business qualification of Originator and Destination.
O.Authen_Age	The TOE must provide for the expiration of authentication information (except for biometric information). This includes cryptographic keys, certificates and commitment data.
O.Authen_Indep	A user may share the same authentication information as another user, but the TOE must not reveal this fact.
O.Authen_Protect	The TOE must protect authentication information.
O.Backup	The TOE must provide data backup and restoration.

Continued on next page

Table A.2 – continued from previous page

Objective Name	Objective Description
O.Confidentiality	The TOE must be able to protect information from unauthorized disclosure. This includes TSF data and user data during transmission. (User data internal to a system is protected by internal access control.)
O.Consistency	The TOE must be able to maintain consistency between TSF data stored on different systems such as user identity, user attributes, and timing information.
O.Dynamic	The TOE must provide for the selection of auditable events and management of the audit trail during normal operations so that suspected unauthorized activity can be monitored as it occurs without alerting a perpetrator.
O.Filter	The TOE must provide for the filtering of communications with respect to resource utilization
O.Integrity	The TOE must be able to protect and verify the integrity of stored or transmitted data. This includes user and TSF data such as commitment data, business data, business data with commitment data, or certificates.
O.Key_Indep_Gen	The TOE must be able to generate new keys independently of broken keys.

Continued on next page

Table A.2 – continued from previous page

Objective Name	Objective Description
O.Key_Indep_Distrib	The TOE must distribute new keys independently of broken keys.
O.Logon_Limit	The TOE must limit the number of logon attempts.
O.Nonrep_Dest	The TOE must protect against repudiation by the Destination.
O.Nonrep_Orig	The TOE must protect against repudiation by the Originator.
O.Recover	The TOE must be able to recover the TOE to a secure state.
O.Flow_Control	The TOE must enforce an information flow control policy at specified boundaries.
O.Replay	The TOE must protect against replay attacks.
O.Resid_Prot	Information stored within the TOE must not be made available to other users upon release
O.Revoke_Cert	The TOE must provide for the revocation of certificates.
O.Sequence	The TOE must ensure that it is possible to determine the correct sequence of events. All audit trails within a TOE must include reliable time stamps that can be correlated each other to determine the sequence of events. Similarly, the TOE must support the sequencing of communication data so that the insertion or deletion of network packets can be detected.

Continued on next page

Table A.2 – continued from previous page

Objective Name	Objective Description
O.Status	It must be possible to determine the status of security relevant TOE parameters.
O.System_Integrity	The TOE must provide procedures shall be provided to verify the software integrity of the TOE.
O.Unique	The TOE must enforce the uniqueness of an original.
O.Resource	The TOE must protect against loss of availability through resource exhaustion.

Table A.3: Threat - Security Objective Mapping

Threat Name	Objective Name
T.Actions_Traced	O.Access_Control O.Anon
T.Blockage	O.Alt_Channel
T.Change_Data	O.Access_Control O.Authen O.Integrity O.Recover O.Sequence O.Status O.System_Integrity
T.Comm_Failure	O.Alt_Channel O.Audit
T.Data_Theft	O.Access_Control O.Authen
T.Deny_Service	O.Authen O.Access_Control O.Resources
T.Deny_Data	O.Nonrep_Orig O.Attest O.Assoc_User_Action
T.Deny_Receipt	O.Attest O.Nonrep_Dest
T.Deny_Submit	O.Attest O.Nonrep_Orig

Continued on next page

Table A.3 – continued from previous page

Threat Name	Objective Name
T.Disaster	O.Backup O.Recover
T.Disclose_Data	O.Access_Control O.Authen O.Flow_Control O.Resid_Prot O.Authen_Protect
T.False_Routing	O.Authen_Address
T.History_Untraceable	O.Consistency O.Sequence
T.Impersonate	O.Assoc_User_Action O.Authen O.Authen_Address O.Integrity O.Replay O.Sequence
T.Indeterminate_Seq	O.Audit O.Sequence
T.Insider	O.Access_Control O.Assoc_User_Action O.Audit O.Authen O.Dynamic
T.Intercept	O.Authen_Age O.Key_Indep_Gen O.Key_Indep_Trans
T.Invalid_Certificate	O.Access_Control O.Authen_Age O.Integrity O.Revoke_Cert
T.Logon_Attack	O.Logon_Limit
T.Outsider	O.Access_Control O.Authen O.Dynamic O.Flow_Control
T.Physical	O.Backup O.Recover
T.Privacy_Violated	O.Access_Control O.Anon O.Authen
T.Refusal	O.Access_Control O.Integrity O.Revoke_Cert
T.Replace_TOE	O.System_Integrity
T.Replay	O.Sequence O.Replay O.Authen_Address
T.Secret_Disclose	O.Authen_Age O.Authen_Indep O.Authen_Protect

Continued on next page

Table A.3 – continued from previous page

Threat Name	Objective Name
T.Service_Denied	O.Authen O.Access_Control O.Flow_Control O.Denial_Sophisticated O.Dynamic O.Alt_Channel
T.TOE_Fail	O.Backup O.Recover
T.Traffic	O.Filter
T.Unique_Copied	O.Unique

APPENDIX B: PROMELA SPECIFICATIONS

Table B.1: Automatically generated PROMELA model

```
mtype = { FIRST,STATE1,STATE2,STATE3,FINAL }
mtype = { MYTESTOPERATION,CREATE,DESTRUCT }
int myTestProperty = 0
chan trigger = [0] of {mtype}
byte state = FIRST

proctype StateManager () {
    byte triggerEvent = MYTESTOPERATION;
    printf ("Inside manager");
    run MyTestClass();
    do
    :: skip;
        if
        :: state == FINAL;
            printf ("State is final, breaking");
            break
        :: state != FINAL;
            if
            :: skip;
                trigger!MYTESTOPERATION;
            :: skip;
                trigger!CREATE;
            :: skip;
                trigger!DESTRUCT;
            fi
        fi
    od;
printf ("Manager finished")
```

```
}

proctype MyTestClass () {
    bit functionResult = 1;
    byte receivedEvent;
    do
    :: skip;
        atomic{
            trigger?receivedEvent;
            if
            :: skip;
                functionResult=0
            :: skip;
                functionResult=1
            fi;
            if
            :: receivedEvent == MYTESTOPERATION;
                printf ("MYTESTOPERATION event received");
                if
                :: state == STATE1;
                    if
:                       : functionResult;
                        myTestProperty++;
                        state=STATE2
                    :: !functionResult;
                        state=STATE3
                    :: else;
                        skip
                    fi
                :: state == STATE2;
                    state=STATE1
```

```
        :: else;
            skip
        fi
    :: receivedEvent == CREATE;
        printf ("CREATE event received");
        if
        :: state == FIRST;
            state=STATE1
        :: else;
            skip
        fi
    :: receivedEvent == DESTRUCT;
        printf ("DESTRUCT event received");
        if
        :: state == STATE3;
            state=FINAL;
            break
        :: else;
            skip
        fi
    fi;
    if
    :: state == FIRST;
        printf ("State is FIRST")
    :: state == STATE1;
        printf ("State is STATE1")
    :: state == STATE2;
        printf ("State is STATE2")
    :: state == STATE3;
        printf ("State is STATE3")
    :: state == FINAL;
```

```
                printf ("State is FINAL")
            fi
        }
    od;
    printf ("MyTestClass finished")
}

init {
    run StateManager()
}
```

Table B.2: Automatically generated PROMELA model

```
mtype = { FIRST,INITIAL,ACCOUNTLOCKED,FINAL,LOGGEDIN }
mtype = { LOGIN,CREATE,DESTRUCT }
int maxLoginAttempt = 3
int failedLoginAttempts = 0
chan trigger = [0] of {mtype}
byte state = FIRST

proctype StateManager () {
   byte triggerEvent = LOGIN;
   printf ("Inside manager");
   run UserAccount();
   do
   :: skip;
      if
      :: state == FINAL;
         printf ("State is final, breaking");
         break
```

```
        :: state != FINAL;
            if
            :: skip;
                trigger!LOGIN;
            :: skip;
                trigger!CREATE;
            :: skip;
                trigger!DESTRUCT;
            fi
        fi
    od;
    printf ("Manager finished")
}

proctype UserAccount () {
    bit functionResult = 1;
    byte receivedEvent;
    do
    :: skip;
        atomic{
            trigger?receivedEvent;
            if
            :: skip;
                functionResult=0
            :: skip;
                functionResult=1
            fi;
            if
            :: receivedEvent == LOGIN;
                printf ("LOGIN event received");
                if
```

```
:: state == INITIAL;
    if
    :: functionResult;
        state=LOGGEDIN
    :: !functionResult;
        failedLoginAttempts++;
        if
        :: maxLoginAttempt ¡ failedLoginAttempts;
            state=ACCOUNTLOCKED
        :: maxLoginAttempt ¿ failedLoginAttempts;
            state=INITIAL
        :: else;
            skip
        fi
    :: else;
        skip
    fi
:: else;
    skip
fi
:: receivedEvent == CREATE;
    printf ("CREATE event received");
    if
    :: state == FIRST;
        state=INITIAL
    :: else;
        skip
    fi
:: receivedEvent == DESTRUCT;
    printf ("DESTRUCT event received");
    if
```

```
            :: state == ACCOUNTLOCKED;
               state=FINAL;
               break
            :: state == LOGGEDIN;
               state=FINAL;
               break
            :: else;
               skip
            fi
         fi;
         if
         :: state == FIRST;
            printf ("State is FIRST")
         :: state == INITIAL;
            printf ("State is INITIAL")
         :: state == ACCOUNTLOCKED;
            printf ("State is ACCOUNTLOCKED")
         :: state == FINAL;
            printf ("State is FINAL")
         :: state == LOGGEDIN;
            printf ("State is LOGGEDIN")
         fi
      }
   od;
   printf ("UserAccount finished")
}
init {
   run StateManager()
}
```

Table B.3: Automatically generated PROMELA model

```
mtype = { SearchCriteria, MusicCatalogItem, UserSecret, CreditCardInfo,
Music, PurchaseReceipt, None, True, False, Integer} ;

typedef messageTransferred{
   int trustLevel = 0;
   mtype message = None;
}

chan userInterfaceChannel = [1] of {messageTransferred};
chan mobileWalletChannel = [0] of {messageTransferred};
chan musicPlayerChannel = [0] of {messageTransferred};
chan musicProviderChannel = [0] of {messageTransferred};
chan paymentProviderChannel = [0] of {messageTransferred};

int userInterfaceTrustLevel = 5;
int mobileWalletTrustLevel = 4;
int musicPlayerTrustLevel = 1;
int musicProviderTrustLevel = 1;
int paymentProviderTrustLevel = 4;

messageTransferred userInterfaceReceived;
messageTransferred mobileWalletReceived;
messageTransferred musicPlayerReceived;
messageTransferred musicProviderReceived;
messageTransferred paymentProviderReceived;

messageTransferred getSearchCriteria;
messageTransferred getSearchCriteriaReturn;
messageTransferred getMusicCatalog;
messageTransferred getMusicCatalogReturn;
```

```
messageTransferred getMusicCatalogItemSelection;
messageTransferred getMusicCatalogItemSelectionReturn;
messageTransferred purchaseMusic;
messageTransferred purchaseMusicReturn;
messageTransferred requestPaymentAuthorization;
messageTransferred requestPaymentAuthorizationReturn;
messageTransferred getUserSecret;
messageTransferred getUserSecretReturn;
messageTransferred getCreditCard;
messageTransferred getCreditCardReturn;
messageTransferred payForGoods;
messageTransferred payForGoodsReturn;
messageTransferred notifyPurchaseComplete;
messageTransferred downloadPurchasedMusic;
messageTransferred downloadPurchasedMusicReturn;
messageTransferred saveMusic;
messageTransferred saveMusicReturn;
messageTransferred getOwnedMusicCatalog;
messageTransferred getOwnedMusicCatalogReturn;
messageTransferred playMusic;
messageTransferred playMusicReturn;

proctype UserInterface() {
   userInterfaceChannel ! getSearchCriteria;
   userInterfaceChannel ? userInterfaceReceived;
   if
      :: (userInterfaceReceived.message == None)
         userInterfaceChannel ! getSearchCriteriaReturn;
   fi
   userInterfaceChannel ? userInterfaceReceived;
   if
```

```
:: (userInterfaceReceived.message == SearchCriteria)
    musicProviderChannel ! getMusicCatalog;
fi
userInterfaceChannel ? userInterfaceReceived;
if
  :: (userInterfaceReceived.message == MusicCatalogItem)
    userInterfaceChannel ! getMusicCatalogItemSelection;
fi
userInterfaceChannel ? userInterfaceReceived;
if
  :: (userInterfaceReceived.message == None)
    userInterfaceChannel ! getMusicCatalogItemSelectionReturn;
fi
userInterfaceChannel ? userInterfaceReceived;
if
  :: (userInterfaceReceived.message == MusicCatalogItem)
    musicProviderChannel ! purchaseMusic;
fi
userInterfaceChannel ? userInterfaceReceived;
if
  :: (userInterfaceReceived.message == True)
fi
userInterfaceChannel ? userInterfaceReceived;
if
  :: (userInterfaceReceived.message == Integer)
    userInterfaceChannel ! getUserSecret;
fi
userInterfaceChannel ? userInterfaceReceived;
if
  :: (userInterfaceReceived.message == None)
    userInterfaceChannel ! getUserSecretReturn;
```

```
fi
userInterfaceChannel ? userInterfaceReceived;
if
   :: (userInterfaceReceived.message == UserSecret)
      mobileWalletChannel ! getCreditCard;
fi
userInterfaceChannel ? userInterfaceReceived;
if
   :: (userInterfaceReceived.message == CreditCardInfo)
      paymentProviderChannel ! payForGoods;
fi
userInterfaceChannel ? userInterfaceReceived;
if
   :: (userInterfaceReceived.message == True)
fi
userInterfaceChannel ? userInterfaceReceived;
if
   :: (userInterfaceReceived.message == PurchaseReceipt)
      musicProviderChannel ! downloadPurchasedMusic;
fi
userInterfaceChannel ? userInterfaceReceived;
if
   :: (userInterfaceReceived.message == Music)
      musicPlayerChannel ! saveMusic;
fi
musicPlayerChannel ! getOwnedMusicCatalog;
userInterfaceChannel ? userInterfaceReceived;
if
   :: (userInterfaceReceived.message == MusicCatalogItem)
      musicPlayerChannel ! playMusic;
fi
```

```
}

proctype MobileWallet() {
    mobileWalletChannel ? mobileWalletReceived;
    if
        :: (mobileWalletReceived.message == UserSecret)
            userInterfaceChannel ! getCreditCardReturn;
    fi
}

proctype MusicPlayer() {
    musicPlayerChannel ? musicPlayerReceived;
    if
        :: (musicPlayerReceived.message == Music)
    fi
    musicPlayerChannel ? musicPlayerReceived;
    if
        :: (musicPlayerReceived.message == SearchCriteria)
            userInterfaceChannel ! getOwnedMusicCatalogReturn;
    fi
    musicPlayerChannel ? musicPlayerReceived;
    if
        :: (musicPlayerReceived.message == MusicCatalogItem)
    fi
}

proctype MusicProvider() {
    musicProviderChannel ? musicProviderReceived;
    if
        :: (musicProviderReceived.message == SearchCriteria)
            userInterfaceChannel ! getMusicCatalogReturn;
```

```
    fi
    musicProviderChannel ? musicProviderReceived;
    if
        :: (musicProviderReceived.message == MusicCatalogItem)
            userInterfaceChannel ! purchaseMusicReturn;
    fi
    userInterfaceChannel ! requestPaymentAuthorization;
    musicProviderChannel ? musicProviderReceived;
    if
        :: (musicProviderReceived.message == PurchaseReceipt)
    fi
    musicProviderChannel ? musicProviderReceived;
    if
        :: (musicProviderReceived.message == MusicCatalogItem)
            userInterfaceChannel ! downloadPurchasedMusicReturn;
    fi
}

proctype PaymentProvider() {
    paymentProviderChannel ? paymentProviderReceived;
    if
        :: (paymentProviderReceived.message == CreditCardInfo)
            userInterfaceChannel ! payForGoodsReturn;
    fi
    musicProviderChannel ! notifyPurchaseComplete;
    userInterfaceChannel ! notifyPurchaseComplete;
}

init {
    getSearchCriteria.message = None;
    getSearchCriteria.trustLevel = 0;
```

getSearchCriteriaReturn.message = SearchCriteria;

getSearchCriteriaReturn.trustLevel = 0;

getMusicCatalog.message = SearchCriteria;

getMusicCatalog.trustLevel = 0;

getMusicCatalogReturn.message = MusicCatalogItem;

getMusicCatalogItemSelection.message = None;

getMusicCatalogItemSelection.trustLevel = 0;

getMusicCatalogItemSelectionReturn.message = MusicCatalogItem;

getMusicCatalogItemSelectionReturn.trustLevel = 0;

purchaseMusic.message = MusicCatalogItem;

purchaseMusic.trustLevel = 0;

purchaseMusicReturn.message = True;

purchaseMusicReturn.trustLevel = 0;

requestPaymentAuthorization.message = Integer;

requestPaymentAuthorization.trustLevel = 0;

getUserSecret.message = None;

getUserSecret.trustLevel = 0;

getUserSecretReturn.message = UserSecret;

getUserSecretReturn.trustLevel = 5;

getCreditCard.message = UserSecret;

getCreditCard.trustLevel = 5;

getCreditCardReturn.message = CreditCardInfo;

getCreditCardReturn.trustLevel = 4;

payForGoods.message = CreditCardInfo;

payForGoods.trustLevel = 4;

payForGoodsReturn.message = True;

payForGoodsReturn.trustLevel = 0;

notifyPurchaseComplete.message = PurchaseReceipt;

notifyPurchaseComplete.trustLevel = 2;

downloadPurchasedMusic.message = MusicCatalogItem;

downloadPurchasedMusic.trustLevel = 0;

```
    downloadPurchasedMusicReturn.message = Music;
    downloadPurchasedMusicReturn.trustLevel = 1;
    saveMusic.message = Music;
    saveMusic.trustLevel = 1;
    getOwnedMusicCatalog.message = SearchCriteria;
    getOwnedMusicCatalog.trustLevel = 0;
    getOwnedMusicCatalogReturn.message = MusicCatalogItem;
    getOwnedMusicCatalogReturn.trustLevel = 0;
    playMusic.message = MusicCatalogItem;
    playMusic.trustLevel = 0;

    run UserInterface();
    run MobileWallet();
    run MusicProvider();
    run MusicPlayer();
    run PaymentProvider();
}
```

Table B.4: Automatically Generated PROMELA Specification For Embedded Function Definitions

```
byte generatedPassword[255]
bool createPasswordComplete
active proctype createPassword () {
c_code {srand(1);
int passwordLength = 6;
int counter = 0;
char c;
for (counter = 0 ; counter ¡ passwordLength; counter++) {
    c = (char) (rand() % 128);
    now.generatedPassword[counter] = c;
}
now.generatedPassword[counter] = '0';};
createPasswordComplete=true
}
```

REFERENCES

1. "OMG UML Specifications", `http://www.omg.org/spec/UML/`, accessed, December, 2015.

2. "UML Profile Diagrams", `http://www.uml-diagrams.org/profile-diagrams.html`, accessed, December, 2015.

3. "OMG OCL Specifications", `http://www.omg.org/spec/OCL/`, accessed, December, 2015.

4. "OMG MDA Specifications", `http://www.omg.org/mda/`, accessed, December, 2015.

5. "Eclipse MDT", `https://eclipse.org/modeling/mdt/`, accessed, December, 2015.

6. Ben-Ari, M., *Principles of the Spin Model Checker*, Springer, 2008.

7. CCRA, "Common Criteria Part 1: Introduction and General Model", `http://www.commoncriteriaportal.org/thecc.html`, accessed, December, 2015.

8. CCRA, "Common Criteria Part 2: Security Functional Requirements", `http://www.commoncriteriaportal.org/thecc.html`, accessed, December, 2015.

9. CCRA, "Common Criteria Part 3: Security Assurance Requirements", http://www.commoncriteriaportal.org/thecc.html, accessed, December, 2015.

10. Ware, M. S., J. B. Bowles and C. M. Eastman, "Using the Common Criteria to Elicit Security Requirements with Use Cases", *Proceedings of IEEE*, pp. 273–278,

March 2005.

11. Beckert, B. and D. Bruns, "Mind the Gap: Formal Verification and the Common Criteria (Discussion Paper)", *VERIFY-2010 (EPiC Series, vol. 3)*, pp. 4–12, 2010.

12. *Software Security: Building Security*, Addison-Wesley, 2006.

13. *The Security Development Life cycle: SDL: A Process for Developing Demonstrably More Secure Software*, Addison-Wesley, 2006.

14. Davis, N., *Secure Software Development Life Cycle Processes*, Tech. rep., National Cyber Security Division, US Department of Homeland Security, 2006.

15. Villarroel, R., E. Fernandez-Medina and E. Piattini, "Secure Information Systems Development; a Survey and Comparison", *Computers and Security*, pp. 308–321, 2005.

16. Cabot, J. and N. Zannone, "Towards an Integrated Framework for Model-driven Security Engineering", *Modeling Security Workshop*, 2008.

17. Smith, S., A. Beaulieu and W. G. Phillips, "Modeling and Verifying Security Protocols Using UML 2", *IEEE International Systems Conference (SysCon)*, 2011.

18. Jurjens, J., "UMLsec: Extending UML for Secure Systems Development", *UML 2002 - The Unified Modeling Language*, Vol. 2460, pp. 412–425, Springer, 2002.

19. Jurjens, J., *Secure Systems Development with UML*, Springer, 2003.

20. Jurjens, J., "Automated Verification of UMLsec Models for Security Requirements", *International Conference on the Unified Modeling Language, No. 7*, Lisbon, Portugal, 2004.

21. Peralta, K. P., A. M. Orozco, A. F. Zorzo and F. M. Oliveira, "Specifying Security Aspects in UML Models", *Proceedings of the Workshop on Modeling Security*, 2008.

22. Goudalo, W. and D. Seret, "Toward the Engineering of Security of Information Systems (ESIS): UML and the IS Confidentiality", *The Second International Conference on Emerging Security Information, Systems and Technologies*, 2008.

23. Villarroel, R., E. Fernandez-Medina, M. Piattini and J. Trujillo, "A UML 2.0/OCL Extension for Designing Secure Data Warehouses", *Journal of Research and Practice in Information Technology*, Vol. 38, No. 1, February 2006.

24. Sohr, K., T. Mustafa, X. Bao and G. J. Ahn, "Enforcing Role-Based Access Control Policies in Web Services with UML and OCL", *Annual Computer Security Applications Conference*, 2008.

25. Pavlich-Mariscal, J., L. Michel and S. Demurjian, "Enhancing UML to Model Custom Security Aspects", *Proceedings of Aspect Oriented Modeling*, 2007.

26. Lodderstedt, T., D. Basin and J. Doser, "SecureUML: A UML-Based Modeling Language for Model-Driven Security", *Lecture notes in computer science*, 2002.

27. Poniszewska-Maranda, A., "Security Constraints in Access Control of Information System Using UML Language", *Proceedings of the 15th IEEE International Workshops on Enabling Technologies:Infrastructure for Collaborative Enterprises*, 2006.

28. Fernandez-Medina, E., M. Piattini and M. A. Serrano, "Specification of Security Constraint in UML", *IEEE 35th International Carnahan Conference on Security Technology*, pp. 163–171, 2001.

29. Mehr, F. and U. Schreirer, "Modelling of Message Security Concerns with UML", *9th International Conference on Enterprise Information Systems*, 2007.

30. Borek, M., N. Moebius, K. Stenzel and W. Reif, "Security Requirements Formalized With OCL in a Model-driven Approach", *International Workshop on Model-Driven Requirements Engineering (MoDRE)*, 2013.

31. Lincke, S. J., T. H. Knautz and M. D. Lowery, "Designing System Security with UML Misuse Deployment Diagrams", *IEEE Sixth International Conference on Software Security and Reliability Companion (SERE-C)*, 2012.

32. Muntean, P., A. Rabbi and C. Eckert, "Automated Detection of Information Flow Vulnerabilities in UML State Charts and C Code", *IEEE International Conference on Software Quality, Reliability and Security - Companion (QRS-C)*, 2015.

33. Katkalov, K., K. Stenzel, M. Borek and W. Reif, "Modeling Information Flow Properties with UML", *7th International Conference on New Technologies, Mobility and Security (NTMS)*, 2015.

34. Buchholtz, M., C. Montangero, L. Perrone and S. Semprini, "For-LySa: UML for Authentication Analysis", *Global Computing: IST/FET International Workshop*, 2004.

35. Peterson, M. J., J. B. Bowles and J. M. Eastman, "UMLpac: An Approach for Integrating Security into UML Class Design", *Proceedings of IEEE*, pp. 267–272, March 2005.

36. Alam, M., M. Hafner and R. Breu, "A Constraint Based Role Based Access Control in the SECTET a Model-driven Approach", *Proceedings of the 2006 International Conference on Privacy, Security and Trust: Bridge the Gap Between PST Technologies and Business Services*, 2006.

37. Breu, R., M. Hafner, F. Innerhofer-Oberperfler and F. Wozak, "Model-Driven Security Engineering of Service Oriented Systems", *Information Systems and eBusiness Technologies*, Vol. 5, pp. 59–71, 2008.

38. Mullins, J. and M. Bergeron, *SOCLe Project: UML and OCL Semantics in ASM*, Tech. rep., National Defense of Canada, December 2004.

39. Charpentier, R. and M. Salois, "Security Modelling for C2IS in UML/OCL", *8th*

International Command and Control Research and Technology Symposium, 2003.

40. Börger, E., A. Cavarra and E. Riccobene, "On Formalizing UML State Machines Using ASM", *Information and Software Technology*, Vol. 46, No. 5, pp. 287–292, 2004.

41. Mezei, G., T. Levendovszky and H. Charaf, "Formalizing the Evaluation of OCL Constraints", *Acta Polytechnica Hungarica*, Vol. 4, No. 1, 2007.

42. Moebius, N., D. Haneberg, W. Reif and G. Schellhorn, "A Modeling Framework for the Development of Provably Secure E-Commerce Applications", *International Conference on Software Engineering Advances*, 2007.

43. Zisman, A., "A Static Verification Framework for Secure Peer-to-Peer Applications", *International Conference on Internet and Web Applications and Services*, 2007.

44. "On-the-fly, LTL Model Checking with SPIN", `http://spinroot.com/spin/whatispin.html`, accessed, December, 2015.

45. Siveroni, I., A. Zisman and G. Spanoudakis, "A UML-based Static Verification Framework for Security", *Requirements Engineering*, , No. 15, pp. 95–118, 2010.

46. Munante, D., L. Gallon and P. Aniorte, "An Approach Based on Model-Driven Engineering to Define Security Policies Using OrBAC", *Eighth International Conference on Availability, Reliability and Security (ARES)*, 2013.

47. Kaliappan, P. S., H. Koenig and V. K. Kaliappan, "Designing and Verifying Communication Protocols Using Model Driven Architecture and Spin Model Checker", *Proceedings of the 2008 International Conference on Computer Science and Software Engineering*, Vol. 2, 2008.

48. Mana, A., J. A. Montenegro, D. Ray, F. Sanchez-Cid and M. I. Yague, "Integrating

and Automating Security Engineering In UM", *IASTED International Conference on Communication, Network and Information Security*, New York, NY, USA, 2003.

49. Basin, D., J. Doser and T. Loddersted, "Model Driven Security: From UML Models to Access Control Infrastructures", *ACM Transactions on Software Engineering and Methodology*, Vol. 15, pp. 39–91, 2006.

50. Mouheb, D., C. Talhi, V. Lima, M. Debbabi, L. Wang and M. Pourzandi, "Weaving Security Aspects into UML 2.0 Design Models", *Aspect Oriented Modelling*, 2009.

51. Borek, M., K. Stenzel, K. Katkalov and W. Reif, "Abstracting Security-critical Applications for Model Checking in a Model-driven Approach", *6th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, 2015.

52. Ledru, Y., A. Idani and J. L. Richier, "Validation of a Security Policy by the Test of Its Formal B Specification – A Case Study", *IEEE/ACM 3rd FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, 2015.

53. Mellado, D., E. Fernandez-Medina and M. Piattini, "A Common Criteria based Security Requirements Engineering Process for The Development of Secure Information Systems", *Computer Standards and Interfaces*, , No. 29, pp. 244–253, 2007.

54. Saeki, M. and H. Kaiya, "Using Common Criteria as Reusable Knowledge in Security Requirements Elicitation", *Modeling Security Workshop, Models 08*, 2008.

55. Morimoto, S. and J. Cheng, "Patterning Protection Profiles by UML for Security Specifications", *Proceedings of the 2005 International Conference on Computational Intelligence for Modeling*, 2005.

56. Beckers, K., D. Hatebur and M. Heisel, "A Problem-Based Threat Analysis in Compliance with Common Criteria", *Eighth International Conference on Availability, Reliability and Security (ARES)*, 2013.

57. Aoki, Y. and S. Matsuura, "Verifying Security Requirements Using Model Checking Technique for UML-based Requirements Specification", *IEEE 1st International Workshop on Requirements Engineering and Testing (RET)*, 2014.

58. Wang, D. and P. Torr, "Checking Threat Modeling Data Flow Diagrams for Implementation Conformance and Security", *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 393–396, 2007.

59. Abi-Antoun, M. and J. M. Barnes, *STRIDE-based Security Model in Acme, Technical Report CMU-ISR-10-106*, Tech. rep., Carnegie Mellon University, 2010.

60. Association, E. C. M., *Extended Commercially Oriented Functionality Class (E-COFC), Standard ECMA-271, ECMA Technical Report TR/78*, Tech. rep., ECMA, 1999.

61. "Concise Promela Reference", `http://spinroot.com/spin/Man/Quick.html`, accessed, December, 2015.