A Formal Engineering Approach for Interweaving Functional and Security Requirements of RESTful Web APIs

Doctoral Dissertation Reviewed by Hosei University

A Formal Engineering Approach for Interweaving Functional and Security Requirements of RESTful Web APIs

by

Busalire Onesmus Emeka

A dissertation submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy(Science)

Graduate School of Computer and Information Sciences

HOSEI UNIVERSITY

March 2023

Supervisor: Professor Soichiro Hidaka

Co-Supervisor: Professor Shaoying Liu

Co-Supervisor: Professor Satoshi Obana

To my family and friends

Acknowledgments

First and foremost, I would like to pass my profound gratitude to my supervisors, Prof. Soichiro Hidaka, Prof. Shaoying Liu and Prof. Satoshi Obana. They have invested a great deal of time and knowledge in providing guidance throughout the duration of this work. Without their support, this thesis would not have materialised. I'd like to thank Prof. Liu whose guidance during my masters program enabled me to continue pursuing my research to the doctoral level. His constant encouragement, positive critique of my work and suggestions on how I can improve my research work helped me maintain a positive outlook and kept pushing on for better results.

I'd also like to pass special appreciations to Prof. Soichiro Hidaka for his constant hands on guidance throughout this work which included in-depth reviews of my conference paper submissions, weekly reviews and guidance of my research progress and his ability to patiently and tirelessly discuss the problems i encountered and providing insights pointing to the target solution.

Many thanks also goes to the support staff, Graduate School of Computer and Information Sciences for their constant assistance in my campus life especially on their friendly way of engagement and providing timely feedback to any set of questions i would pose at them. I sincerely also thank my Japanese class teacher for her efforts in helping me improve my Japanese level.

Finally, very profound gratitude goes to my parents for believing in my grit to pursue a doctorate program and their constant words of encouragement throughout my study period.

Table of Contents

List of Tables

List of Figures

Abstract

RESTful Web API adoption has become ubiquitous with the proliferation of REST APIs in almost all domains with modern web applications embracing the micro-service architecture. This vibrant and expanding adoption of APIs, has made an increasing amount of data to be funneled through systems which require proper access management to ensure that web assets are secured. A RESTful API provides data using the HTTP protocol over the network, interacting with databases and other services and must preserve its security properties. Currently, practitioners are facing two major challenges for developing high quality secure RESTful APIs. One, REST is not a protocol. Instead, it is a set of guidelines that define how web resources can be designed and accessed over HTTP endpoints. There are a set of guidelines which stipulate how related resources should be structured using hierarchical URIs as well as how specific well-defined actions on those resources should be represented using different HTTP verbs. Whereas security has always been critical in the design of RESTful APIs, there are no clear formal models utilizing a secure-by-design approach that interweaves both the functional and security requirements. The other challenge is how to effectively utilize a model driven approach for constructing precise requirements and design specifications so that the security of a RESTful API is considered as a concern that transcends across functionality rather than individual isolated operations.

This thesis proposes a novel technique that encourages a model driven approach to specifying and verifying APIs functional and security requirements with the practical formal method SOFL (Structured-Object-Oriented Formal Language). Our proposed approach provides a generic 6 step model driven approach for designing security aware APIs by utilizing concepts of domain models, domain primitives, Ecore metamodel and SOFL. The first step involves generating a flat file with APIs resource listings. In this step, we extract resource definitions from an input RESTful API documentation written in RAML using an existing RAML parser. The output of this step is a flat file representing API resources as defined in the RAML input file. This step is fully automated. The second step involves automatic construction of an API resource graph that will work as a blue print for creating the target API domain model. The input for this step is the flat file generated from step 1 and the output is a directed graph (digraph) of API resource. We leverage on an algorithm which we created that takes a list of lists of API resource nodes and the defined API root resource node as an input, and constructs a digraph highlighting all the API resources as an output. In step 3, we use the generated digraph as a guide to manually define the API's initial domain model as the target output with an aggregate root corresponding to the root node of the input digraph and the rest of the nodes corresponding to domain model entities. In actual sense, the generated digraph in step 2 is a barebone representation of the target domain model, but what is missing in the domain model at this stage in the distinction between

containment and reference relationship between entities. The resulting domain model describes the entire ecosystem of the modeled API in the form of Domain Driven Design Concepts of aggregates, aggregate root, entities, entity relationships, value objects and aggregate boundaries. The fourth step, which takes our newly defined domain model as input, involves a threat modeling process using Attack Defense Trees (ADTrees) to identify potential security vulnerabilities in our API domain model and their countermeasures. Countermeasures that can enforce secure constructs on the attributes and behavior of their associated domain entities are modeled as domain primitives. Domain primitives are distilled versions of value objects with proper invariants. These invariants enforce security constraints on the behavior of their associated entities in our API domain model. The output of this step is a complete refined domain model with additional security invariants from the threat modeling process defined as domain primitives in the refined domain model. This fourth step achieves our first interweaving of functional and security requirements in an implicit manner. The fifth step involves creating an Ecore metamodel that describes the structure of our API domain model. In this step, we rely on the refined domain model as input and create an Ecore metamodel that our refined domain model corresponds to, as an output. Specifically, this step encompasses structural modeling of our target RESTful API. The structural model describes the possible resource types, their attributes, and relations as well as their interface and representations. The sixth and the final step involves behavioral modeling. The input for this step is an Ecore metamodel from step 5 and the output is formal security aware RESTful API specifications in SOFL language. Our goal here is to define RESTful API behaviors that consist of actions corresponding to their respective HTTP verbs i.e., GET, POST, PUT, DELETE and PATCH. For example, *CreateAction* creates a new resource, an *UpdateAction* provides the capability to change the value of attributes and *ReturnAction* allows for response definition including the Representation and all metadata. To achieve behavioral modelling, we transform our API methods into SOFL processes. We take advantage of the expressive nature of SOFL processes to define our modeled API behaviors. We achieve the interweaving of functional and security requirements by injecting boolean formulas in post condition of SOFL processes. To verify whether the interweaved functional and security requirements implement all expected functions correctly and satisfy the desired security constraints, we can optionally perform specification testing. Since implicit specifications do not indicate algorithms for implementation but are rather expressed with predicate expressions involving pre and post conditions for any given specification, we can substitute all the variables involved a process with concrete values of their types with results and evaluate their results in the form of truth values *true* or *false*. When conducting specification testing, we apply SOFL process animation technique to obtain the set of concrete values of output variables for each process functional scenario. We analyse test results by comparing the evaluation results with an analysis criteria. An analysis criteria is a predicate expression representing the properties to be verified. If the evaluation results are consistent with

the predicate expression, the analysis show consistency between the process specification and its associated requirement. We generate the test cases for both input and output variables based on the user requirements. The test cases generated are usually based on test targets which are predicate expressions, such as the pre and post conditions of a process. when testing for conformance of a process specification to its associated service operation, we only need to observe the execution results of the process by providing concrete input values to all of its functional scenarios and analyze their defining conditions relative to user requirements. We present an empirical case study for validating the practicality and usability of our model driven formal engineering approach by applying it in developing a Salon Booking System. A total of 32 services covering functionalities provided by the Salon Booking System API were developed. We defined process specifications for the API services with their respective security requirements. The security requirements were injected in the threat modeling and behavioral modeling phase of our approach. We test for the interweaving of functional and security requirements in the specifications generated by our approach by conducting tests relative to original RAML specifications. Failed tests were exhibited in cases where injected security measure like requirement of an object level access control is not respected i.e., object level access control is not checked. Our generated SOFL specification correctly rejects such case by returning an appropriate error message while the original RAML specification incorrectly dictates to accept such request, because it is not aware of such measure. We further demonstrate a technique for generating SOFL specifications from a domain model via model to text transformation. The model to text transformation technique semi-automates the generation of SOFL formal specification in step 6 of our proposed approach. The technique allows for isolation of dynamic and static sections of the generated specifications. This enables our technique to have the capability of preserving the static sections of the target specifications while updating the dynamic sections in response to the changes of the underlying domain model representing the RESTful API in design. Specifically, our contribution is provision of a systemic model driven formal engineering approach for design and development of secure RESTful web APIs. The proposed approach offers a six-step methodology covering both structural and behavioral modelling of APIs with a focus on security. The most distinguished merit of the model to text transformation is the utilization of the API's domain model as well as a metamodel that the domain model corresponds to as the foundation for generation of formal SOFL specifications that is a representation of API's functional and security requirements.

Chapter1

Introduction

1.1 Background

1.1.1 Web APIs

A web API (Application Programming Interface) is a set of functions and procedures that allow clients access and build upon the data and functionality of an existing application available over the web through the HTTP protocol [1]. Clients should only know about the interface and nothing about the intricacies of its implementation. A given interface can have multiple implementations, and a client written against the interface can switch between implementations seamlessly without any overheads. To invoke a web API remotely, you need to have a protocol defined for inter-process communication. Examples of some protocols that facilitate interprocess communication include CORBA, .NET Remoting, Java RMI, REST (over HTTP) and SOAP. Java RMI provides the infrastructure-level support to invoke a Java API remotely from a nonlocal Java virtual machine. The RMI infrastructure at the client side serializes all the requests from the client into the wire and deserializes into Java objects at the server side by its RMI infrastructure [2]. Java RMI is language dependent and can only be invoked by a Java client.

A SOAP-based web service provides mechanisms to build and invoke a remote API in a language and platform agnostic manner. It utilizes an XML payload to pass a message from one end to the other. SOAP has a defined structure, and there exist a large number of specifications that offer a guide in defining its structure. A typical SOAP specification defines a request/response protocol between a client and a server. Web Services Description Language (WSDL) specification defines the way you describe a SOAP service. The security of a SOAP based service are defined by WS-Security, WS-Trust, and WS-Federation specifications. WS-Policy provides a framework to build quality-of-service expressions around SOAP services. WS-Security Policy defines the security requirements of a SOAP service in a standard way, built on top of the WS-Policy framework [2]. Contrary to SOAP, REpresentational State Transfer (REST) offers a design paradigm rather than a set of design rules and protocols. Many web APIs nowadays adopt REST[3] architectural style which allows building loosely coupled API designs relying on HTTP and the web friendly JSON data representation format. The loosely coupling approach makes client applications have flexibility and reusability of an API in terms of the fact that its elements can be easily added, replaced and changed.

1.1.2 Web APIs in Practice

Web APIs rely on HTTP to facilitate the communication between the host of the API also known as the *provider* and the system making an API request also referred to as the *consumer*. By using HTTP, web APIs can take advantage of the protocol's standardized methods, status codes, and client server relationships, allowing developers to write code that can automatically handle data. An API endpoint, defined as a URL for interacting with part of an API, serves requests from an API consumer. Below are some examples of API endpoints from a salon booking service.

http://salonapi.com/api/salons

http://salonapi.com/api/salons/{salon_id}

http://salonapi.com/api/salons/customers

http://salonapi.com/api/salons/{salon_id}/profile

The data being requested by an API is referred to as a *Resource*. A singleton resource is a unique object, such as `http://salonapi.com/api/salons/{salon_id}`. A group of resources, such as `http://salonapi.com/api/salons` is referred to as a *collection*, while a *subcollection* refers to a collection within a particular resource. For example, `http://salonapi.com/api/salons/{salon_id}/profile` is the endpoint to access the profile subcollection of a specific salon.

In the event a consumer requests a resource from a provider, the request passes through an *API gateway* [4]. An API management acts as an entry point to a web application. The API gateway filters bad requests, monitors incoming traffic, and routes each request to the proper service or *microservice* [4]. A *microservice* is a modular segment of a web app dedicated to handle a specific function and they use APIs to transfer data and trigger actions. An API gateway can also handle additional functionalities such as security controls like authorization, authentication, encryption of data in transit using SSL [5], load balancing and API rate limiting.

1.1.3 RESTful Web APIs

The concept of REST was introduced by Roy Fielding in his PhD dissertation, "Architectural Styles and the Design of Network-based Software Architectures"[3]. REST relies on HTTP protocol for data communication and revolves around the concept of resources where each and every component is considered as a resource. These resources are accessed via a common interface using HTTP methods such as GET for retrieving a resource, PUT for updating a resource, POST for creating a resource and DELETE for removing a resource. Contrary to other web services, REST is an architectural style and protocol agnostic.

When describing the REST design paradigm, there are two essential concepts: *resources* and *representations*. A *resource* is anything that can be referenced as an item by itself [1]. It is an artifact that can be stored on a computer such as an electronic document, a row in a database, or the result of running an algorithm. A *representation* or a resource representation is a document response that a server sends when a web browser sends an HTTP request for a resource. Every resource must have a Uniform Resource Identifier (URI). The URI uniquely identifies a resource therefore when a client makes an HTTP request to a URL, it gets a representation of the underlying resource and makes it capable of being manipulated using an application protocol such as HTTP. A resource can have more than one URI but a URI identifies only one resource. A resource's URIs may provide different information about the location of the resource or the protocol that can be used to manipulate it.

RESTful web APIs design depends on six constraints as defined by Fielding [3]. These constraints essentially provide a set of guidelines for an HTTP resource-based architecture:

- **Uniform Interface**: RESTful APIs should have a uniform interface. It is agnostic to any requesting client thus all clients should be able to access a server in the same way.
- **Client-Server Architecture**: RESTful APIs should conform to a client-server architecture where the clients consume the requested information, and servers provide the requested information.
- **Stateless**: The latest HTTP request made by a client is not stored in the server thus each and every request is treated as new. If a client needs a stateful user operation such as requiring the user to log in once then perform other authorized operations, then the client should supply all the necessary information such as authorization tokens and headers for each request.
- **Cacheability**: *Caching* provides a mechanism of increasing request throughput by storing commonly requested data on the client side or in a server cache. A response from an API that conforms to REST guidelines should indicate whether the response is cacheable. The mechanism that describes the implementation of a cacheable property is as follows. When a client makes a request for information, it firsts check its local storage for the target information. If it doesn't find the information, it passes the request to the server, which checks its local storage for the requested information. If the data is not there either, the request could be passed to other servers, such as database servers, where the data can be retrieved.
- **Layered System**: The client should be able to request data from an endpoint without the knowledge of the underlying server architecture implementation.

- **Code on demand**: This guideline allows for code to be sent to the client for execution. Code on demand establishes a technology coupling between web servers and their clients, since the client must be able to to understand and execute the code that downloads on-demand from the server.

REST APIs have common headers similar to HTTP headers [6]. These include:

- **Authorization** headers pass a token or credentials to an API provider. They take the following format `Authorization: <type> <token/credentials>`. The different type of authorization headers include *Basic, Bearer Token, Digest Auth, OAUTH and API Key. Basic* offers a simple authentication scheme built into the HTTP protocol. The client sends HTTP requests with an Authorization header containing the word *Basic* and a base64-encoded (non-encrypted) string containing a username and a password. *Bearer Token* uses an API token, a cryptic string, usually generated by the server in response to a login request. *Digest Auth* communicates credentials in an encrypted form. It applies a hash algorithm to the provided username and password. The password is converted to response and then sent to the server. *OAUTH* permits client applications to access data provided by a third-party API without exposing your login details. *API Key* authorization relies on a token that a client provides when making API calls. A key-value pair is normally sent to the API either in the request headers or query parameters.

- **Content-Type** headers indicate the type of media being transferred. Common *Content-Type* headers for RESTful APIs include: **application/json** for specifying JavaScript Object Notation (JSON) [7] as a media type, **application/xml** for specifying XML as a media type, and **application/x-www-form-urlencoded** which specifies a format in which the values being sent are encoded and separated by an ampersand (&), and an equal sign (=) is used between key/value pairs.

- **Middleware (X) Headers** take the format of `X-<header>` and serve different purposes. For example, `X-Response-Time` as an API response indicates how long a response took to process. `X-API-Key` can be used as an authorization header for API keys. `X-Powered-By` can be used to provide additional information about the interfacing backend services and `X-Rate-Limit` can be used to tell the consumer how many requests they can make within a given time frame.

1.1.4 RESTful Web APIs Vulnerabilities

In this section, we shall give an introductory overview of common API security vulnerabilities as documented by the Open Web Application Security Project (OWASP) [8]. The mitigation strategies to the highlighted

vulnerabilities will provide a set of security requirements which our proposed framework seeks to interweave with the functional requirements as discussed in chapter 5.

1.1.4.1 Information Disclosure

A deployed API may service both private and public users. When an API share sensitive information with unprivileged users, it exhibits an information disclosure vulnerability. Sensitive data can include any information that attackers can leverage to their advantage such as a list of usernames and their access control levels which attackers can use techniques such as brute-force, credentials-stuffing, or password-spraying attacks. alternatively, an API can exhibit information disclosure vulnerability through verbose error messaging which can reveal sensitive information about resources, users and the API's underlying architecture.

1.1.4.2 Broken Object Level Authorization

Broken Object Level Authorization (BOLA) [9] vulnerabilities occur when an API consumer accesses API resources they are not authorized to access. This occurs if an API endpoint does not have object-level access controls. Missing object-level access control means the API won't perform checks to make sure users can only access their own resources. Attackers can leverage on BOLA vulnerabilities by propagating an attack against an API endpoint via fuzzing parameters in an API's URL path and sorting through the results to determine the existence of BOLA vulnerabilities.

1.1.4.3 Broken User Authentication

The stateless constraint of RESTful APIs requires users to undergo a registration process in order to acquire a unique access token. Users are then required to include this token in all of their future requests to affirm that they are authorized to make such requests. The registration process used to obtain an API token, or the token handling process could exhibit some weaknesses. Token handling process could be the storage of tokens, the method of transmitting tokens across a network, or the presence of hardcoded tokens in JavaScript source files. Attackers can use a captured token to gain access to an API endpoint or bypass authentication.

1.1.4.4 Excessive Data Exposure

Excessive data exposure vulnerability occurs when an API endpoint responds with more information than is needed to fulfill a request. A typical example is when an API consumer requests information for their user account and receives information about other user accounts as well.

1.1.4.5 Broken Function Level Authorization

APIs may expose capability functionalities of different user roles in the interfacing underlying system. API providers will often provide different types of accounts such as public, private, administrators etc. Broken Function Level Authorization (BFLA) occurs where a user of one role or group is able to access the API functionality of another role or group. BFLA can be a lateral move, where an attacker or user gains usage of the functions of a similarly privileged group, or it could be a privilege escalation, where an attacker or a user is able to use the functions of a more privileged group. BFLA vulnerabilities occur when API's endpoints access controls are not implemented correctly. Sometimes, an API won't always use administrative endpoints for administrative functionality. Instead, the functionality could be based on HTTP request methods such as GET, POST, PUT, and DELETE. If a provider doesn't restrict the HTTP methods a consumer can use, simply making an unauthorized request with a different method could indicate a BFLA vulnerability.

1.1.4.6 Mass Assignment

This API vulnerability occurs when an API consumer includes more parameters in their requests than the application intended and the application adds these parameters to code variables or internal objects. With this scenario, a consumer may be able to edit object properties or escalate privileges. Mass assignment vulnerabilities are mainly a result of an API endpoint failing to sanitize its request input correctly or in entirety. A lack of input sanitization also gives attackers capabilities of uploading malicious payloads thus propagating other types of attacks like Cross Site Scripting (XSS attacks).

1.1.4.7 Injection Flaws

These occur when a request is passed to the API's supporting infrastructure and the API provider doesn't filter the input to remove unwanted characters. The infrastructure might treat data from the request as code and run it against its target environment. Existence of this vulnerability gives room for attackers to conduct injection attacks such as SQL injection, NoSQL injection, and system command injection. In each of the aforementioned injection attacks, the API delivers your unsanitized payload directly to the operating system running the application or its database. As a result, if you send a payload containing SQL commands to a vulnerable API that uses a SQL database, the API will pass the commands to the database, which will process and perform the commands. SQL Injection flaws can be executed either manually by submitting metacharacters as input to the API or using automated solution like SQLmap [10]. Metacharacters are charaters that SQL treats as functions rather than data e.g. `--` is a metacharacter that tells the SQL interpreter to ignore the input that follows because it is a comment. A null byte like *;%00* could cause a

verbose SQL-related error to be sent as a response resulting in Information Disclosure vulnerability.

1.1.5 Requirements Engineering and Formal Methods

Formal methods which provide techniques such as formal specification, refinement, and verification have contributed significantly to the evolution of software engineering. The concept of using pre- and post-conditions have been adopted in some programming languages to support the principle of "design by contract" [11]. Over the recent past formal methods have seen adoption in real software projects [12] [13] [14], especially safety critical applications. In its strict sense, formal methods adopts use of mathematical approaches and notations in specification, design, analysis and assurance of computer systems and software [15]. Formal methods offers a stepwise refinement approach in software development. The first step involves formalizing the customer's informal requirements into formal specifications. To check for conformance between the formal specification and its equivalent representation of its original informal requirements, validation approaches such as specification animation [16] [17] [18] [19] have been proposed. Inconsistencies between the formal specifications and its equivalent informal specifications are ironed out through a refinement process guided by the validation approaches. The refinement is an iterative process which focuses on a gradual but ultimate process of transforming the refined formal specifications into an executable program. Each refinement process requires a formal verification process to be performed to ensure consistency between the refined specifications and the final executable program or a refined specifications and its predecessor. Even though formal verification offer powerful techniques for checking consistency between the delivered systems and their specifications, it still remains too expensive to be deployed in most software projects. To effectively support the application of formal methods to the requirements engineering process in software development, *formal engineering* [15] [20] methods are proposed.

1.1.6 Formal Engineering Methods

Formal engineering methods form a bridge between formal methods and their application, providing techniques for incorporating formal methods into the entire software engineering process [15]. They provide specifications languages that not only offer textual notations in certain formal languages but also integrate graphical notations. The specifications include intuitive diagrams and precise textual formal specifications integrated in a coherent manner to describe the overall system architecture and its functions. In addition, formal engineering methods provide effective mechanisms and techniques that aid practitioners in constructing formal specifications allowing them to effectively apply formal methods in practice.

Formal engineering methods also adopt rigorous but practical techniques for verifying and validating

specifications and their equivalent executable programs. They support evolution rather than strict refinement in specifications to programs transformations. The concept of evolution of specification means that the construction of specifications and their equivalent programs transformed from the specifications do not necessarily have to satisfy the strict refinement rules.

One of the most famous formal engineering method is SOFL (Structured Object-Oriented Formal Language) method [15] [21] [22]. It offers methods and supporting techniques for specification construction, specification transformation, and system verification and validation. Our research presented in this dissertation relies on the basic principles of the SOFL formal engineering method as one of the core ingredients that makes it achieve its overarching goal of interweaving functional and security requirements of RESTful web APIs.

1.1.7 Web API Security Requirements Specifications

When designing web API, its paramount to also consider the environment in which the API will operate as well as the potential threats that may be leveled against the API in that environment. Threats define ways in which a security goal may be violated with regards to any of the assets or resources provided by a deployed API. To define mitigating measures against API attacks, one needs to consider realistic threats against an API then zero in efforts on areas where you can identify gaps in your set of API defenses against a set of threats. Many a times, Web API's are self-documenting, which means they can provide information such as their internal structure and implementation of their business logic. As such, activities for systematically identifying threats to an API so that they can be recorded, tracked and mitigated must be put in place when drafting APIs security requirements specifications. These activities are collectively referred to as threat modeling.

Threat modeling [23] refers to the process analyzing a system to look for weaknesses that come from less-desirable design choices. The goal of the activity is to identify these weaknesses before they are subconsciously included in the system so you can take corrective action as early as possible. Threat modelling involves analysing a system as a collection of its components and their interaction with the outside world and the actors that may perform actions on these systems. In system design process, threat modelling activities offer the benefit of building cleaner architectures, well defined system trust boundaries and better documentation. There are several techniques proposed for conducting API threat modeling as described here shortly afterwards. However, the general process adopts the following steps:

- First, draw a system diagram that showcase the main logical components of your target API
- Next, focus on identifying the trust boundaries between parts of the system with the rule of thumb

here being everything within a trust boundary should be managed by the same owner.

- After defining trust boundaries, draw arrows that highlight data flows between the different parts of the system and across trust boundaries.
- Analyze each component of data flows within the system. While doing the analysis try to identify threats that may compromise your API security goals while paying key attention to data flow across trust boundaries.
- Finalize the threat modelling activity by recording threats to ensure they are tracked and managed.

It is worth noting that the goal of threat modeling is to identify general API threats rather than enumerating every possible attack that can be leveraged against your API. Modeling an API for threat analysis can be done by adopting one or a combination of the following model types [23]:

- **Data flow diagrams** - Data flow diagrams (DFDs) [24] describe the flow of data among components in a system and the properties of each component and flow. They provide a visual way of describing an abstracted system. Data flow diagrams often represent a system in layers where each layer indicates a level of abstraction. The top layer i.e. *context layer* represents the system interactions with external entities such as remote systems or users. Subsequent layers drill down into more details on individual system components and interactions, until the target level of details is achieved. DFDs leverage standard shapes that represent a process or operating unit within a system under consideration.
- **Sequence diagrams** - Sequence diagrams model an event based sequence of actions, providing a context about the way a modelled system behaves under any temporal aspects required for detailed analysis. Sequence diagrams show the order of operations used in a system communication flow, revealing important information such as which actor initiated the communication and any steps in the process that may introduce a security or privacy risk. This makes it easier in finding flaws in business logic and protocol handling and can also highlight critical design failures such as system areas that lack exceptional handling or areas where security controls are not consistently implemented.
- **Process flow diagrams** - Process flow diagrams [24] show the sequence and the directional flow of operations through a system, revealing the the activity chain of events in a system rather than the flow of specific messages and component state transitions. Process flow diagrams can be used to complement sequence diagrams in threat modelling. For example, an activity chain from a process flow diagram can be described by a sequence diagram using labels that indicate which segments of a system's message flow are bound to a specific event.

- **Fishbone diagrams** - Fishbone diagrams [25] also referred to as *Ishikawa* diagrams, are used for root cause analysis of a problem statement thus can help in identifying weaknesses in a system for any given area. They offer a modelling process that can help understand the chain of events that can lead to exploitation of a weakness. Construction of a fishbone diagram involves first defining the effect you want to model, then identifying a set of primary causes that lead to the defined effect and finally identifying the set of causes that drive the primary cause. For example, if we want to model API excessive data exposure, we first define the main effect as *API excessive data exposure*. Next, we identify a set of primary causes that lead to API excessive data exposure. We can identify absence of data filtering and overly verbose error messages. Finally, we identify the set of causes that drive the primary causes. For absence of data filtering, we can identify causes such as poor access control implementation, broken object access authorization and an insider threat (malicious developer). For overly verbose error messages, we can identify causes such as Improper API configuration and improper error handling. Figure 1.1 shows an example of a fishbone diagram in a complete state with the expected effect, primary and secondary causes.

- **Attack Trees** - Attack trees [26] offer an attacker-centric threat modelling approach. An attack tree model is a rooted tree with the root node representing the goal or the desired threat modelling outcome. Each node is labeled with an action to be taken. Construction of an attack tree involves two steps i.e. identifying a target or goal of an attack and identifying actions to be taken to achieve the target or goal. Attack trees provide the foundation of another threat modelling approach, Attack Defense Trees (ADTree) [27] which give attention to both attack strategies that can compromise a target system and their respective defense strategies.

- **Attack Defense Trees (ADTrees)** - Proposed by Kordy et al. [27], ADTrees focus on providing a threat modeling approach that describes measures or paths an attacker might use to compromise a system and mitigating measures against these attack paths as defense mechanisms. We describe the concept of ADTrees in details in Sec. 5.2. We adopt ADTrees as our threat modeling activity of our choice due to its capability providing both attack scenarios that can be leveraged on our target API as well as mitigation measures that counter these attacks, which we infuse in our API specifications as security requirements.

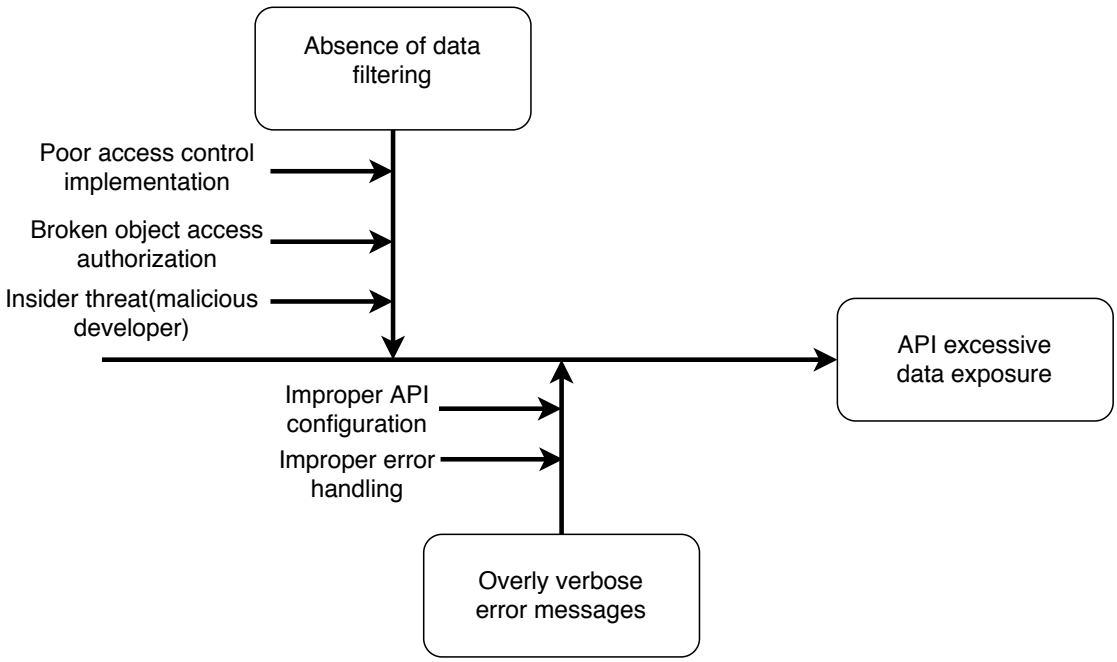

Figure 1.1: Sample complete fishbone diagram modelling API excessive data exposure threat

1.2 Research Motivation

Increased popularity of RESTful web APIs [28] [29] has led to a demand for engineering methods for developing reliable, high quality and secure web APIs. In general, APIs development follows the conventional engineering process of requirement analysis, API design, API code generation, API testing and maintenance. In the domain of software engineering, the stages of requirements analysis and API design are usually merged and referred to as the modelling phase. In this phase, the API business requirements are collected, analyzed and elicited as specifications. The API design stage focuses on the clarification of the expected API business functions and the design of the API architecture for implementing target API endpoints.

The expected API business functions and API architecture are implemented by a programming language of choice in the coding stage. Once the code activity is finalized, API testing kicks off. The testing process focuses on the logical internals of the target API, ensuring that the API performs the expected functions as its specification defines. The specification and the delivered API provide a foundation for API maintenance.

Although modelling and testing techniques for functional and security requirements are critical in API design, effective modelling that provide an interweaving approach for eliciting APIs functional and security requirements are still great challenges in developing secure RESTful web APIs in practice.

1. One major challenge is that REST is a design paradigm and protocol-agnostic. It does not rely on any

set of defined standards to describe the implementation of a RESTful API. This poses a challenge in the development and testing for satisfiability of a RESTful API property such as security. Since RESTful APIs expose internal business services and data to a set of public and/or private heterogeneous client applications, the level of security offered by these APIs must be extremely high, since their breach may cause huge financial and business integrity losses on the part of the service providers. An ideal and secure REST API must exactly and accurately function as intended and preserve its security properties during its operation. If a RESTful API is to provide access to some exposed business data to a requesting client application, it must fulfill the functional requirement of the client application without violation of its' security properties. It must also not violate the security properties of the system providing the data. Moreover, while each individual operation in an API may be secure on its own, combinations of operations might not be. Therefore, adopting a model that enables the capture and verification of error free functional and security requirements of an API is crucial in the development of an API.

2. The other challenge is how to effectively utilize a requirement engineering model that offers both implicit and explicit elicitation of API security requirements. Implicit by conducting model refinement and defining model constraints that offer coverage against API threats and vulnerabilities, and explicit by directly applying threat modeling techniques on a model using already documented API threats and vulnerabilities.

Formal methods have been proven to offer an approach to the construction and verification for precise, consistent and correct specifications using mathematical notations. Research reveals that formal methods have been effective in capturing requirements, identifying errors and transforming specifications to programs [15]. However, in practice, there exist limitations in applying formal methods like VDM [30], B-Method and Z notation because they require high skills for abstraction and their notations offer a steep learning curve for most engineers in the industry. In addition, their formal proof techniques and refinements are difficult and expensive to apply in practice.

1.3 Proposed Solution

In this thesis, we propose a new practical model-driven approach that addresses the aforementioned challenges. We present an approach based on the existing Domain Driven Design [31] that utilizes a metamodel offering a strict foundation for what an API does. The model is strict in the mathematical sense of being precise and exact that its concepts, attributes, behaviors and relations are unambiguous. The model relies

on domain primitives [32] which combine secure constructs to define the smallest building blocks of a domain, and utilizes Ecore [33] metamodel that defines the abstract syntax, the possible domain elements, and their relations in between. We adopt Attack Defense Trees [27] as our threat model of choice for identifying countermeasures to RESTful API security vulnerabilities.

Specifically, we have made the following new contributions. We propose, a new model-driven approach for interweaving functional and security requirements based on Domain Driven Design principles, Ecore metamodel and the expressive nature of SOFL process definition to describe the behaviour of our modeled API. This encourages resolving security issues both implicitly and explicitly. Implicitly by applying strict invariants on domain primitives [32], and explicitly by applying Attack Defense Trees (ADTrees) [27] to model API threats and identify common documented API vulnerabilities and their associated countermeasures. By focusing on the domain and domain primitives, many security bugs can be solved implicitly. For example, applying a strict invariant defined as a domain primitive on an API's POST input not only protects the API endpoint against injection attacks but also ensures the true meaning of the input is captured. Therefore, any malicious input not satisfying the definition is rejected and the API endpoint becomes more secure.

1.4 Summary

In this chapter, we have first introduced the background of web and RESTful APIs. We have pointed out the major challenges in designing security aware RESTful web APIs and illustrated the motivation of our research. We have also presented our proposed solution to tackle these challenges and the main contributions of our research work. In the next chapter, we will discuss about the different RESTful API schemas giving details of the role they play in API design.

Chapter2

API Schema Model

2.1 Introduction

A schema model provides a contract describing what an API is, how it works, and what the end points are going to be. A schema model ensures that everyone has a shared understanding of what an API will do and how each resource will be represented in a complete API. There are two most popular schema modeling systems and the markup languages used for API design:

- RESTful API Modeling Language (RAML) [34], is in YAML format that also supports JSON and Markdown [35]
- OpenAPI, which also supports Markdown, is a JSON [7] object which may be represented either in JSON orYAML[36]

A schema model is made up of resources and resource methods. A resource can be defined as a single object or list within a system (collection). A resource has to have at least one Uniform Resource Identifier (URI). A URI describes the name and address of a resource. In a strict sense, no two resources are the same. However, at some moment in time, two different resources may point to the same data. In RESTful web services, a resource responds to one or more of the six standard HTTP methods:

- GET - Retrieve the representation of the object or list
- PUT - Replace (update) the object
- POST - Create a new entry at the top level
- DELETE - Delete an object from the system
- HEAD – Retrieve metadata for an object
- OPTIONS – Check which HTTP methods a particular resource supports

Every resource is expected to expose the same uniform interface with similarities in the way they work. An object's value can be obtained by invoking a GET request to that object's URI. If you are interested in only getting the metadata for an object, then you can send a HEAD request to the same URI. To create a

resource, you utilize a PUT request sent to a URI that incorporates the resource's name. Adding an object to a resource (collection), can be accomplished by sending a PUT to a URI that incorporates the resource name and object name. Sending a DELETE request to a URI that points to a resource or an object deletes the target object, resource or resource collection.

2.2 RESTful Web API Domain Modelling Language (RAML)

RESTful API modelling Language (RAML) provides concise, expressive language for describing RESTful APIs. Like a functional specification, an RAML document describes how the API will behave. RAML is a non-proprietary, vendor neutral open specification [34]. RAML uses markdown, in human readable description parts. In this section, we shall briefly explain the structure of an RAML document. RAML uses Yet Another Mark up Language (YAML) as its underlying specification format. As such, all nodes such as keys, values, and tags, in RAML specifications are case-sensitive.

Listing 2.1 below shows a sample structure of an RAML document

```
#%RAML 1.0
title: Salon API
version: 1
baseUri: http://localhost:8000/api/{version}
mediaType: application/json

uses:
  shapes: ./dataTypes/shapes.raml

traits:
  contentCacheable: !include ./traits/content-cacheable.raml
  secured: !include ./traits/secured.raml
  pageable: !include  ./traits/pageable.raml

resourceTypes:
  collection: !include ./resourceTypes/collection.raml
  booking: !include ./resourceTypes/booking.raml
  salon: !include ./resourceTypes/salon.raml
  customer: !include ./resourceTypes/customer.raml
  stylist: !include ./resourceTypes/salon.raml
  service_category: !include ./resourceTypes/service-category.raml
  service: !include ./resourceTypes/services.raml

securedBy: [oauth_2_0]
securitySchemes:
  oauth_2_0: !include ./securitySchemes/oauth_2_0.raml

/salons:
```

```
29 type:  { collection: {response-type : shapes.SalonData[], request-type : shapes.NewSalonRequestData } }
30 get:
31   is: [pageable]
32   responses:
33     200:
34       body:
35         application/json:
36           example:|
37             {
38               "salons": [
39               {
40                 "id": "6ddfty-078rty-8986533gh",
41                 "business_name": "Salon Star"
42               },
43               {
44                 "id": "4567-dfgty-4456789ft",
45                 "business_name": "Happy Paradise"
46               }
47               {
48                 "id": "t678-3456-6789hgyt",
49                 "business_name": "Women Heaven"
50               }
51               ]
52             }
53
54 post:
55   is: [secured]
56   description: |
57     salon data created correctly for a salon
58   queryParameters:
59     access_token: string
60     business_name: string
61     business_type: string
62     business_description: string
63     business_phone_number: string
64     business_email: string
65   body:
66     type: shapes.NewSalonRequestData
67   responses:
68     200:
69       body:
70         application/json:
71           example: |
72             {"message": " New Salon created successfully!"}
73           ...
```

Listing 2.1: Sample RAML document

The following table enumerates the nodes at the root of an RAML document (Line 1 to Line 26).

Node	Description
#%RAML 1.0 (*required*)	Indicates which markup language is used. If other modeling compatible with RAML is used, its specified here. This is always formatted as a comment, and any other comments in the document will start with the same #% markup.
title (*required*)	Tells a reader what the API is designed to do.
baseURI (*required*)	Defines the base URL for all endpoints defined by the API. It must be the same for all endpoints.
version	When the version is included as part of the baseUri, the specification is tied to the specified version on this line.
mediaType	Declares the default media types to use for request and response bodies (payloads), for example "application/json" .
traits	Declares traits for use within the API. A trait provides method-level nodes such as a description, headers, query string parameters, and responses.
resourceTypes	They define methods and other nodes. A resource that uses a resource type has an inheritance relationship with its nodes. A resource type can inherit from, another resource type. The relationship between resource types and resources is defined through an inheritance chain pattern.
securedBy	Declares the security schemes that apply to every resource and method in the API. The mechanisms for secure data access, identification of requests and determination of access control policies as well as data objects visibility are defined by the security schemes.
securitySchemes	Declare applicable security schemes to be applied and used within the API.

Table 2.1: Nodes at the root of sample RAML document

More details about RAML specification can be inferred from here [34].

2.3 OpenAPI Specification

OpenAPI [37] was one of the earliest schema modeling frameworks available for web API design. An openAPI document is a document or set of documents that defines or describes an API. OpenAPI is a JSON onject which may be represented either in JSON or YAML. An OpenAPI definition uses and conforms to the OpenAPI Specification. The root resource in OpenAPI requires different information that contained in

an RAML document. Listing 2.2 below shows a sample of an OpenAPI specification focussing on its root elements. Table 2.2 describes the specific pieces of the listed root section.

```
swagger: '2.0'
# basic metadata about API
info:
    title: Salon API
    version: 1
# the host to call and interact with this server
host: localhost:8000/api/
# the base path which is appended to the host
basePath: /v1

schemes:
    -https

consumes:
    -application/json

produces:
    -application/json

paths:
  /salons
  /salons/{salonid}

  ...

/salons:
    get:
        description: Retrieves a list of salons from the system
        parameters:
         - description: String to math in the salon name
           in: query
           name: nameString
           required: false
           type: string
        responses:
         '200':
           description: Salon list
           schema:
            type: array
            items:
             type: object
             properties:
              id:
                type: string
```

```
45            business_name:
46              type: string
47        examples:
48          application/json:
49            [
50                {
51                  "id": "6ddfty-078rty-8986533gh",
52                  "business_name": "Salon Star"
53                },
54                {
55                  "id": "4567-dfgty-4456789ft",
56                  "business_name": "Happy Paradise"
57                }
58                {
59                  "id": "t678-3456-6789hgyt",
60                  "business_name": "Women Heaven"
61                }
62            ]
63            ...
```

Listing 2.2: Sample OpenAPI document

Root section piece	**Description**
swagger (*required*)	Indicates the version of OpenAPI being used
info (*required*)	A block of information related to the API description, with the following required fields. **title (required)** - Represents the tile of the documented API.**version (required)** - Indicates the version of your OpenAPI tool.
host (*optional*)	Indicates the host only. If the host isn't included, the system hosting the documentation is implied.
basepath (*optional*)	The base path for all API endpoints. This should start with a /. If not included, it will be expected that the API is served directly under the host's root.
schemes (*optional*)	The scheme (such as http://) that describes how the API can be accessed. If not included, it will be set to the same scheme used to access the documentation.
consumes/produces (*optional*)	These parameters indicate the content-type sent for responses and accepted in requests.
paths (*required*)	This is a list of the paths that will be served by the API. This is a part of the main OpenAPI object, and the methods, parameters, and behaviors for these paths will be included in the objects for each endpoint.

Table 2.2: Root elements of an OpenAPI Specification Document

More details of OpenAPI specification can be found here [37]. As with RAML, OpenAPI has a couple

of tools and resources for generating OpenAPI schema compliant models. The OpenAPI framework [38] supported by SmartBear, an API testing company, is one of the widely used tools for creating OpenAPI schema models.

2.4 Summary

In this chapter, we have described two of the most common schema modelling languages in API design. Any of the two schema models offer powerful tools in guiding the development of an API to meet its intended use case. However, OpenAPI offers a very strong schema modelling language for defining what is expected of a system which is useful for testing and creating code snippets for a set of web APIs, while RAML is designed to support a design-first development flow and focuses on consistency. Moreover, OpenAPI is best suited to document an existing API rather than designing an API from scratch. On the other hand, RAML has evolved to support API design in a succinct human-centric language. As a matter of choice since our proposed framework is focused on secure API design, we adopt the RAML in defining our source input of our proposed framework. In the following chapter, we shall describe about RESTful Web API Security Requirements Specifications.

Chapter3

Initial Work

3.1 Introduction

Software systems are becoming ubiquitous with software applications being used in the fields of finance, education, transport and logistics etc. With this widespread use of software applications, the security of the data handled and stored by these applications has become more and more important. This has made software security to be one of the most crucial and necessary features of high integrity software systems that support high-risk industries such as the financial service. However, most software engineering methodologies have a bias of taking the standard approach of analysis, design and implementation of software system without considering security, and then add security as an afterthought [39]. A review of recent research in software security reveal that such approach leads to a reasonable number of security vulnerabilities that are usually identified after system implementation and deployment. Fixing such vulnerabilities calls for a "patching" approach since the cost associated with redevelopment of the system at such a point may be too high. However, the "patching" approach is not an acceptable standard in the development of high-risk software systems. Moreover, a study conducted by Hoo, et al. [40], revealed that introducing security analysis in the early stage of the system development cycle can reduce costs related to software development and maintenance from 12-21%. In our publications [41], [42] we proposed security requirements engineering frameworks, which can holistically integrate functional and security requirements of a system software, and eventually yield software requirements that satisfy the required security requirements.

3.2 Initial Work - Concurrent Generation of Software's Functional and Security Requirements

To integrate security attributes into a system software, we proposed a three-step process that promotes a systematic integration of security requirements into the software design process. The methodology works by integrating functional requirements written in Structured Object Oriented Formal Language (SOFL) and standard security requirements drawn from the Common Criteria for Information Technology Security Evaluation [43] and the AICPA's generally accepted privacy principles [44]. First, we expressed the system software's functional requirements in Structured Object Oriented Formal Language (SOFL), including the

construction of Conditional Dataflow Diagrams (CDFDs), giving a graphical denotation of all the data flows, data stores and processes involved in the system software. In the second step, we generated the standard security requirements using SQUARE methodology [45]. SQUARE offers a comprehensive methodology for security requirement engineering. It aims at integrating Security Requirements Engineering into the mainstream software development processes [46].

The SQUARE methodology consists of nine elaborate steps, which provide a means for electing, categorizing, and prioritizing security requirements for a system software and related applications. The outcome of the SQUARE methodology is a set of standard security requirements, broadly be classified into: Identification requirements, Authentication requirements, Authorization requirements, Security auditing requirements, Confidentiality requirements, Integrity requirements, Availability requirements, Non-repudiation requirements, Immunity requirements, Survivability requirements, System maintenance security requirements and Privacy requirements. We integrate these standard security requirements with the functional requirements by expressing those which apply globally to the system software as SOFL module invariants, while those which are only applicable to a few functional processes are expressed as guard conditions of their respective functional processes.

Our final step involves identifying vulnerabilities and threats in the functional requirements provided by our system software. This step has a set of sub steps that include transforming the CDFD we generated in our first step into a process tree. The process tree has the root node representing the top level CDFD process while the parent/child nodes represent decomposed processes from the root node or a parent node. The process tree offers the benefit of a bounded scope, enabling the traversal of all the application's processes from the root node to the forked child processes at the sub-nodes and end-nodes. We then traverse through the nodes of the process tree, and conduct an attack tree analysis at each of the process node. Our goal here is to identify potential vulnerabilities. We define the mitigation strategies of the identified vulnerabilities as additional security requirements, which we integrate with the functional requirements either as SOFL module invariants or as guard conditions of their associated processes. Figure 3.1 gives an overview of the conceptual framework of the technique we proposed.

Our proposed approach in this initial work pushes for addressing the following issues: 1.) Availing to the developer a variety of security methods and their tradeoffs. 2.) Providing a systemic integration of security requirements into the software design. This methodology advocates for a security aware software development process, which is a combination of a selected standard software development methodology, formal methods techniques, and standard security functions [47]. As a precursor and a foundation of our current works on secure design of web APIs, we also conducted research [48] focusing on a formal approach to secure design of RESTful web APIs using Structured Object Oriented Formal Langauge (SOFL) [15]. We

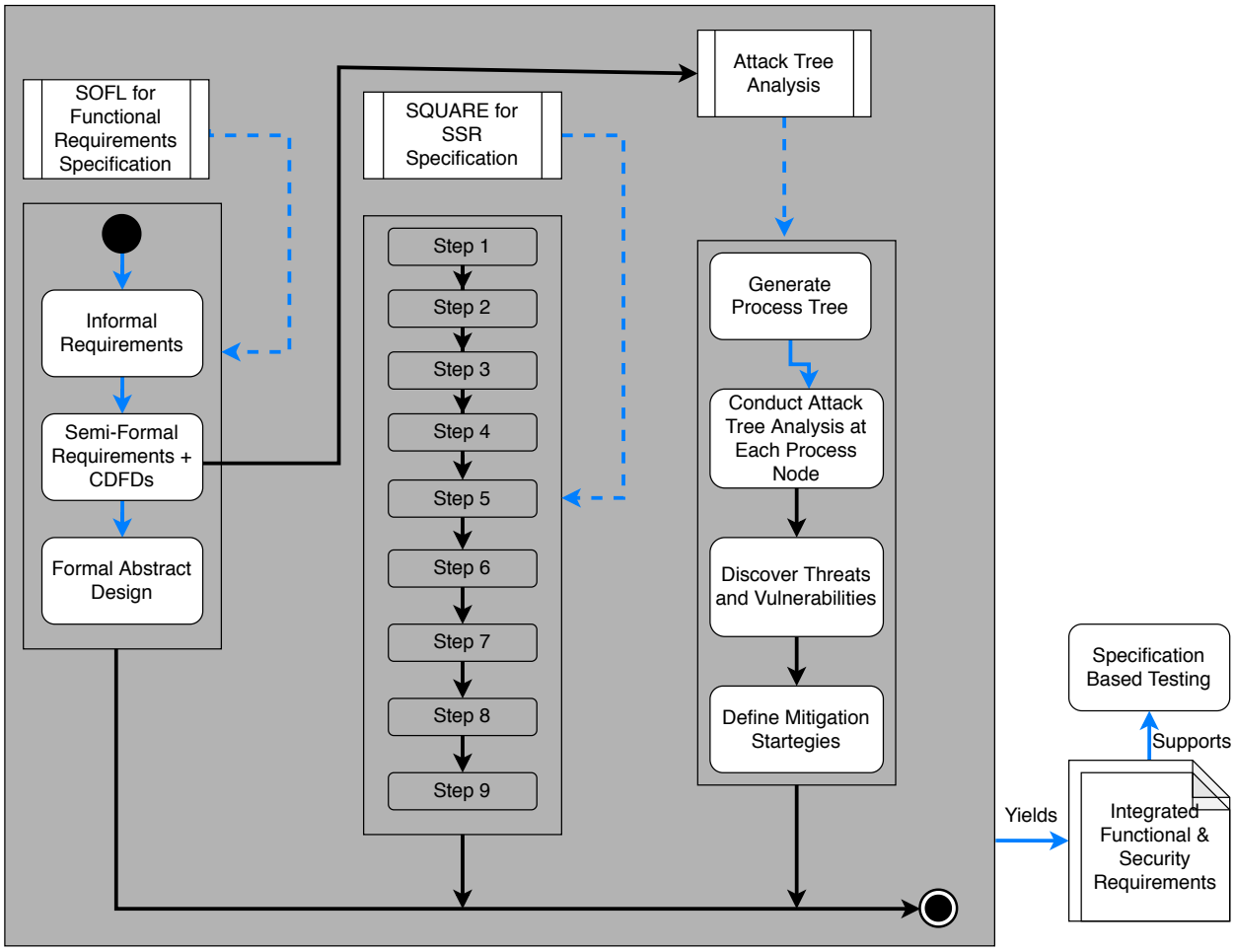

Figure 3.1: A conceptual framework of the proposed technique

shall describe this in detail in the following section.

3.3 Initial Work - A Formal Approach to Secure Design of RESTful Web APIs Using SOFL

This initial work was inspired by the recent trends in wide adoption of REST architectural style in the design of web APIs, which allows building loosely coupled API designs relying on HTTP and the web friendly JSON data representation format. The loosely coupling approach makes client applications have flexibility and re-usability of an API in terms of the fact that its elements can be easily added, replaced and changed. However, REST is a design paradigm and protocol-agnostic. It does not rely on any set of defined standards to describe the implementation of a RESTful API. This poses a challenge in the development and testing for satisfiability of a RESTful API property such as security. Since REST APIs expose internal business services and data to a set of public and/or private heterogeneous client applications, the level of security offered by these APIs must be extremely high, since their breach may cause huge financial and business integrity losses on the part of the service providers.

An ideal and secure REST API must exactly and accurately function as intended and preserve its security properties during its operation. If a REST API is to provide access to some exposed business data to a requesting client application, it must fulfill the functional requirement of the client application without violation of its security properties. It must also not violate the security properties of the system providing the data. In addition, while RESTful services can easily be invoked through a web browser or a client application, it is still difficult for users to fully understand and evaluate their functions with respect to the requirements in the context of target systems, because a few formal descriptions are provided with these services. Moreover, while each individual operation in an API may be secure on its own, combinations of operations might not be. Therefore, adopting a model that enables the capture and verification of precisely defined functional and security requirements of an API is crucial in the development of an API.

We therefore proposed a model offering a formal practical approach to specify and verify security and functional requirements of RESTful APIs using SOFL. Our proposed approach focused on ensuring that all of the expected functional behaviors provided by an API and their related security requirements are captured correctly. This is to ensure that a secure web API delivers both of its business functions and at the same time preserve its security features and that of the system it is interfacing. To achieve this, we construct SOFL formal but comprehensible functional and security requirement specifications from an API description written in RESTful API modelling Language. Our approach offered 3 steps, relying on RAML definitions as

a base for informally describing the APIs functional and security requirements. The first step yields a set of informal functional and their related security requirements expressed in RAML security schemes definitions. These security requirements are defined as constraints upon their associated functional requirements. They express the APIs security goals in operational terms. We derived the informal API security requirements taking into consideration:

- The resources exposed by the API that are to be protected.
- The security goals that are important such as confidentiality of target API resources.
- The mechanisms that are available to achieve these goals such as authentication, access control, audit logging and rate limiting.
- The set of threats relevant to the API. We identified these threats using S.T.R.I.D.E [49] threat modelling process, by analysing the flow of requests/responses across trust boundaries defined by the main logical components of the API, and the target environment for deployment.

The second step focused on transformation of RAML definitions to SOFL based semi-formal specifications by first modelling the API's behavioral features using SOFL's Conditional Data Flow Diagrams (CDFDs) and then express the REST specific request methods as SOFL module processes with informal pre-post conditions. Their related security requirements are defined as either SOFL module invariants or guard condition in the process' pre-post conditions.

While transforming RAML definitions to SOFL semi-formal specifications, we adopt the following rules:

- *Rule 1*: Transform REST request definitions GET, POST, PUT and DELETE to SOFL processes specified with informal pre-post conditions.
- *Rule 2*: Construct CDFDs [50] for the textual semi-formal specifications so that the requirements can be visualized. The CDFDs describe the API's request input and output data flows.
- *Rule 3*: Define REST request parameters as inputs and their types to their respective SOFL processes, and responses as outputs of their associated SOFL processes. All data stores interacting with the inputs and outputs are also defined.
- *Rule 4*: Express the defined RAML security schemes (i.e. security requirements) as SOFL module invariants or guard conditions in the post-condition of the relevant processes. This rule achieves the interweaving of functional and security requirements.

The final step involved formalization of the pre-post conditions and conducting specifications testing to prove the satisfiability of the defined API's functional and security requirements. The resulting formal specifications can then be transformed into executable API code manually or semi-automatically with the help of a supporting tool. We conducted a small experiment to validate our approach by using it to model specifications of a RESTful online banking API. The model specifications included 1 module containing 15 processes with each process reflecting a resource to be retrieved by an API via requests made by a client application. Our approach helped in modeling security requirements in the 15 API processes which we tested for their conformance via specification based conformance testing techniques and confirm the reflection of their expected behaviour in the case study implementation done using Django REST framework. This proposed approach offered benefits such as providing a lightweight security aware model of guiding the design of APIs and generating their equivalent SOFL formal specifications which can be used to verify the conformance of the APIs functional and security requirements. Unlike API specifications expressed in RAML where security requirements are just presented as annotations, our proposed approach encourages proactive analysis of resources exposed by an API end point and how its associated security requirements restrict the abuse of those resources by consuming clients. In other words, our proposed approach encouraged behavioural analysis of client interaction with an API resource from a security point of view in addition to annotating the security requirements that are expected to be satisfied by the client consuming a resource. However, this approach had a few limitations. First, requirement engineers had to be versed with the concept of API security so as to define the required API security requirements as described in step 1 of our proposed approach. The risk posed by this is that, the depth of critical analysis and definition of security requirements would only be as good as the depth of security knowledge of the engineer generating the API specifications. Second, the generated target SOFL formal specifications were not language agnostic. In as much as SOFL can offer many advantages in specification elicitation, it may not be the common preferred choice for engineers generating specifications thus a specification language agnostic approach would have been more ideal. These gaps provided inspirations as well as foundation for our current approach as described in sec. 6.

3.4 Summary

In this chapter, we discussed about our initial works which also focused on security and functional requirements elicitation of software systems. We gave a brief description of the techniques we proposed and highlighted their advantages as well as their contributions in providing a formal engineering approach for both RESTful API and software systems requirements specifications. In the next chapter, we discuss about related research works.

Chapter4

Related Works

4.1 Related Works

Several other models targeting integration of software security attributes at the requirements levels have been proposed by other researchers in the domain of software security requirements engineering. Security requirements engineering is motivated by the fact that analyzing security in the early stage of the system development cycle can significantly save costs for later system development and maintenance [40].

UMLsec [51], an extension of Unified Modelling Language works by expressing security relevant information within UML diagrams. It encapsulates knowledge, making it available to developers in form of a widely used design notation. In addition, UML provides a formal evaluation to check if the constraints associated with the UMLsec requirements are fulfilled in a given specification. The extensions are suggested in form of UML stereotypes, tags, and constraints that can be adopted and applied in various UML diagrams such as activity diagrams as well as sequence diagrams. The stereotypes and tags provide an encapsulation of the recurrent knowledge of security requirements such as secrecy, fair exchange, and secure communication link that can apply in distributed object-oriented systems. UMLSec key concept of addressing security issues lies in the assignment activity of attaching a stereotype or tag to part of a model and retrieving potential threats to the target system. This allows for the behavior of the subsystem to be analyzed and checked for impacts of a successful execution of the threats.

Mouratidis et al. [52] proposed the Secure Tropos methodology which is based on the principle that security should be given focus from the early stages of software development process, and not retrofitted late in the design process or pursued in parallel but separately from functional requirements. The Secure Tropos modelling language works on the principle of a security constraint, a restriction that can influence the analysis and design of a system software under development by; conflicting with some of the requirements of the system, by restricting some alternative design solutions or by conflicting with some of the system requirements. Often, constraints, secure dependencies, threats, security goals, tasks and resources are integrated in the specifications of existing textual descriptions. The secure entities are tagged with an “s” to indicate those tasks, goals and soft goals are security related.

In particular, security requirements can be viewed as constraints on the system software functionalities. In [53], security concerns are integrated througout the phases of Secure Tropos agent-oriented methodology.

This is from early, late requirements, all the way to architecture and detailed design. At the preliminary early requirements phase, a Security Diagram is constructed where the stakeholders review the constraints imposed by the security requirements. In the late requirements phase, security constraints are analyzed against the system-to-be in the Security Diagram. The presentation of the system is portrayed with one or more new actors, with a set of dependencies tied to other actors of the organization. During the last phase i.e. the architectural design stage, secure capabilities, constraints and entities that the new actors introduce are assigned to each agent of the system. This approach adopts a systematic refinement construct, where security experts formulate goals at different levels of abstraction, ranging from high-level, strategic concerns to low-level, technical concerns.

Lamsweerde [54] works on KAOS and introduces the notions of obstacle and anti-goal [55] to analyze security requirements of a system. Undesired states of affairs that prevent stakeholder's goals from being satisfied are captured as KAOS obstacles, whereas anti-goals analyze obstacles that are intentionally imposed by attackers. KAOS utilizes formal methods to systematically refine (security) goals that are specified in Linear Temporal Logic (LTL), using a set of predefined refinement patterns [56]. After discovering specific threats via anti-goal refinement, security requirements (i.e., countermeasures to the threats) are elicited accordingly.

Misuse cases focus on representing behavior(s) that represent an abuse to the resources or services to be offered by the system to be developed. Misuse cases are initiated by misusers. A use case diagram in Fig. 4.1 contains both, use cases and actors, as well as misuse cases and misusers for an online banking system. The use cases are denoted by solid green arrows, misuse by solid red arrows, and solid blue arrows denote system actions that include, extends or prevents actors use cases or misuse cases. Employing misuse cases in security requirements definition allows for the identification of security attacks and associated security requirements during application development. Whittle et al. [57] present a formalized representation of misuse cases that offers intuitive way of analyzing, interpreting and presenting a misuse case model. Misuse cases have gained popularity in representing security concerns in the early stages of software development as they represent aspect of both problems and solutions. However, their limitation lies on their analysis of security requirements and attacks specifically through use case specification. As such, the completeness of coverage with regards to analysis of the security requirements analyzed via use case specification is not guaranteed since this approach may not exhaustively provide coverage against other potential exploitation scenarios that could be leveraged against the system.

As far as the security and modeling of web API's is concerned, several approaches have been done in the field of developing RESTful applications, but to the best of our knowledge, there are a few results that provide detailed model driven techniques with a focus on paying attention to both APIs functional and

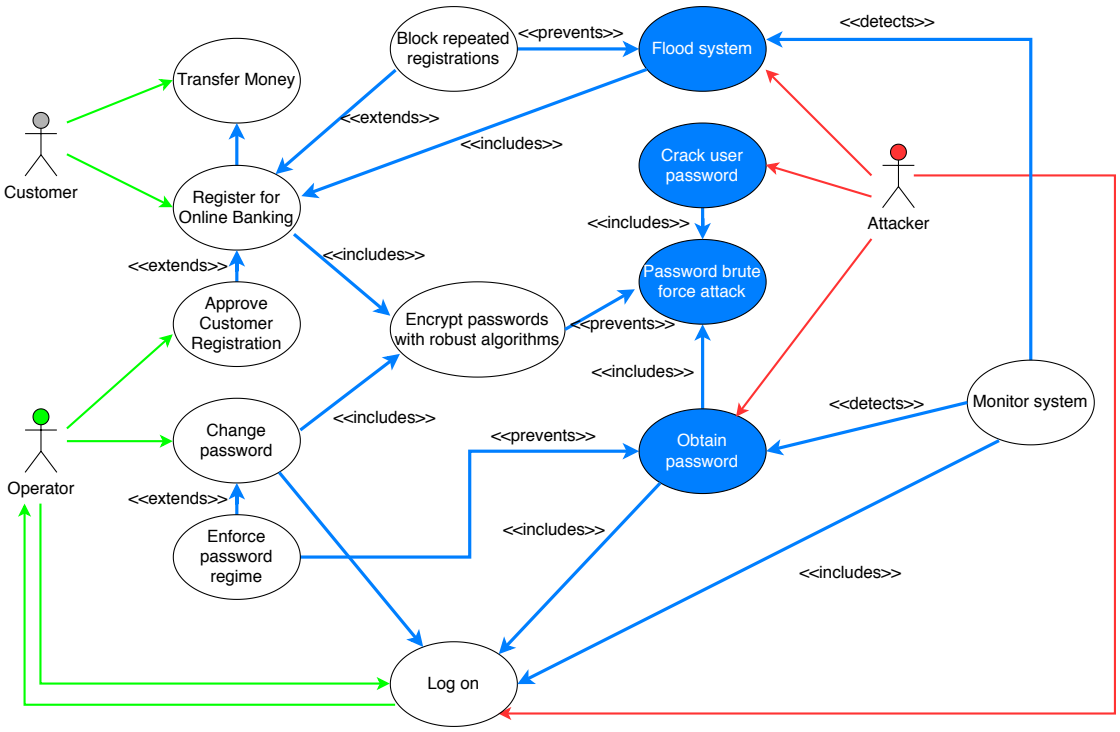

Figure 4.1: A Use case diagram with misuse case and misusers for an Online Banking System

security requirements at the same time. Fett et al. [58] propose a rigorous, systematic formal analysis of OpenID Financial-grade API (FAPI) based on a web infrastructure model. They first develop a precise model of the FAPI in the web infrastructure model, including different profiles for read-only and read-write access, different types of clients, and different combinations of security features, and use their model of FAPI to precisely define central security properties of an API. However, their model treat API security and functional requirements independently. Kopecky et al. [59] present hRESTS as a promising solution for providing a microformat model for RESTful services. However, their approach focuses more on documentation and service discovery with limited clarity on the relationship between API's security and functional requirements and their interdependence. Alqahatni et al. [60] introduce an approach for automatically tracing source code vulnerabilities at the API level across project boundaries. Their approach takes advantage of Semantic Web and its technology stack to establish a unified knowledge representation that can link and analyze vulnerabilities across project boundaries. However, they focus at the source code level rather than the design level. Klien et al. [61] provide an approach for showcasing how the constraints of REST and RESTful HTTP can be precisely formulated within temporal logic. However, their focus is mainly on formal characterization of REST for automated analysis.

Raufet al. [62] propose an approach offering a formal verification technique using Event-B for development

of REST services. Their approach focuses on achieving the correct system architecture through refinement and consistency verification of REST design models. Their model construct behavioral REST service interfaces which provide information on the specific methods that the service can invoke as well as a definition of the pre- and post conditions provided by these methods. Their adoption of Event-B to represent their design models encouraged scalability of their proposed framework by promoting a correct-by-construction development approach and formal verification by proof of theorems. Their approach also supports partial code generation which creates code skeletons of REST services with pre- and post-conditions method definition. However, their approach mainly focused on addressing inconsistency issue in design and resolving state explosion which may arise when dealing with a large number of resources.

4.2 Summary

In this chapter, we have discussed related researches. Compared with other research results, our proposed model driven approach has some distinct merits in addressing security requirements of APIs both implicitly and explicitly while at the same time provide a mechanism for interweaving the functional and security requirements. We aim to achieve this by ensuring that all of the expected functional behaviors provided by an API and their related security requirements must be captured correctly, since a secure web API is expected both to deliver its business functions and to preserve its security features and that of the system it is interfacing. To achieve this, we construct SOFL formal but comprehensible functional and security requirement specifications from an API description written in RESTful API modelling Language (RAML) [63]. Nevertheless, the development of our model driven formal engineering approach is also inspired by the ideas and technologies presented by these related works. In the next chapter, we shall discuss about the building blocks of our proposed approach.

Chapter5

Concepts Used in Proposed Approach Buiding Blocks

In this chapter, we describe the foundational concepts that our proposed approach build on with a focus on interweaving APIs' functional and their respective security requirements. Our proposed approach builds on concepts of Domain Driven Design, threat modeling with ADTrees, Ecore metamodels and SOFL. We describe these concepts in details so as to enable the reader get accustomed to terms, notations and concepts discussed in chapter 5.

5.1 Domain Driven Design (DDD)

Domain models provide unambiguous, strict foundation of what a system should do [31]. This by extension provides a powerful tool that defines what a system should not do. When modeling and implementing that model as requirements specifications, it is crucial to have some building blocks. Domain models are usually based on value objects and entities with larger structures being presented through aggregates. An aggregate is a conceptual boundary that you can use to group parts of the model together allowing you to treat an aggregate as a unit during state changes. The boundary is not arbitrarily chosen but rather it is carefully selected based on deep insights of the model. In order to express a domain model in specifications, you need a set of building blocks which are entities, value objects and aggregates. Figure 5.1 showcase a sample domain model for a salon booking system.

Every part of a domain model has certain characteristics and certain meaning. Entities are model objects that have some distinct properties, unique identifiers and are responsible for coordination of the objects they own, not only to provide cohesion but also to maintain their internal invariants. Let's take an example of an Online Salon Booking System (OSBS), where we have a salon class with attributes such as salon name, address, phone number, establishment data. Every instance of entities has a unique identifier. The ability to identify information in a precise manner as well as coordinating and controlling behavior plays an integral role in preventing security bugs from sneaking into specifications. Thanks to this uniqueness, we can distinguish two instances of the salon class that has the same name, and even have all the same attribute values, by their identifiers, even if they can be interchangeable with each other. Entities are often made up of other model objects. Some attributes and behaviors can be moved out of the entity itself and put into other objects thereby becoming value objects or domain primitives. Value objects have no identity that defines them but rather, they are defined by their values, they are immutable i.e. they describe some

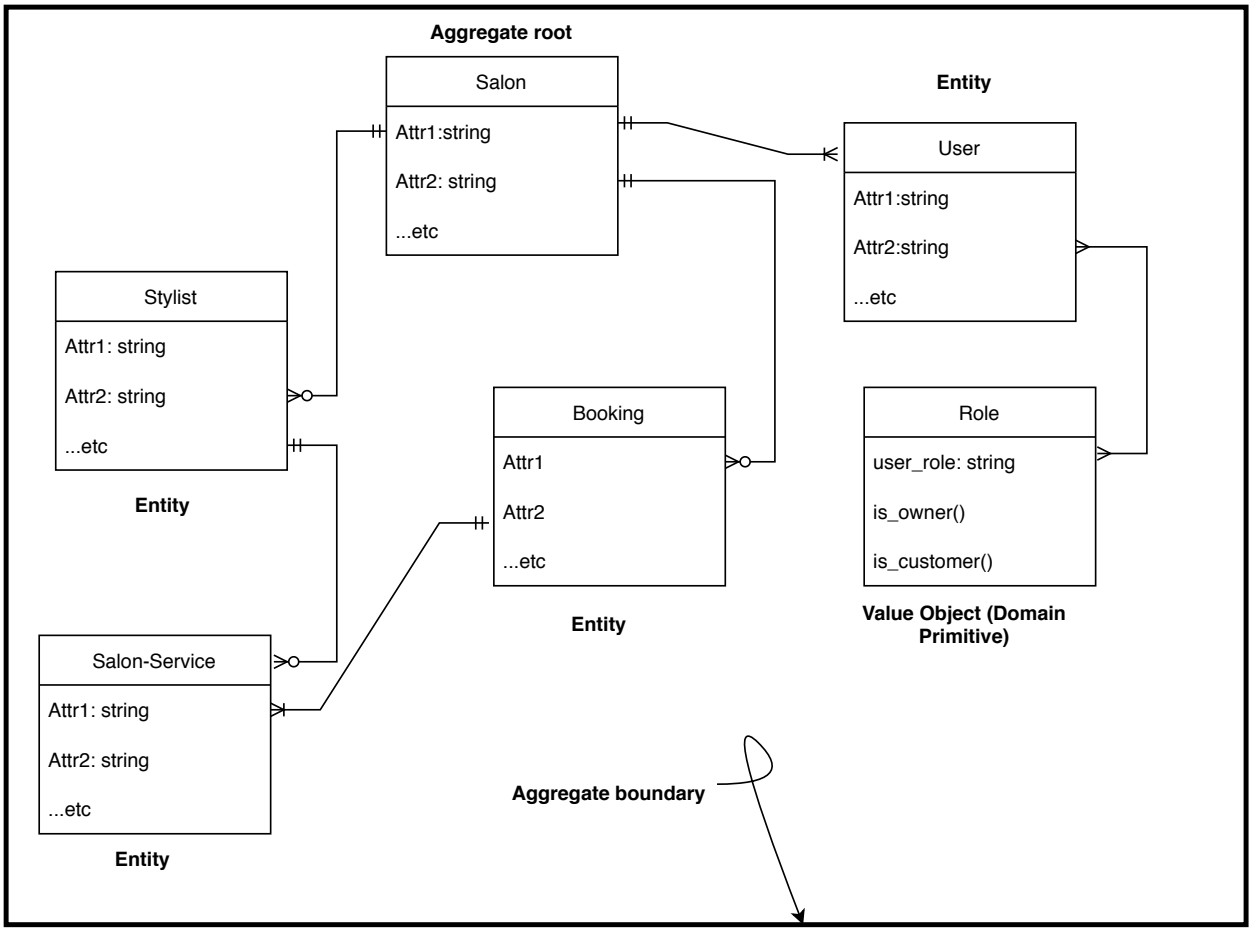

Figure 5.1: Online Salon Booking System Domain Model

attribute or some characteristics but carries no concept of identity, they can reference entities, they explicitly define and enforce important constraints and can be used as attributes of entities and other value objects. *Domain primitives* are distilled versions of value objects with proper invariants. These invariants enforce security constraints on the behavior of their associated entities in our API domain model. Figure 5.1 shows the relationships among aggregates, entities, value objects and domain primitives in DDD using a salon service booking system example. The entity *Salon* represents the aggregate root. The *Salon* entity has a containment relationship with the entity *Booking* which in turn has a containment relationship with the entity *Salon-Service*. The *Role* value object has a referential relationship with the entity *User*. It is a defense mechanism output of an ADTree analysis on the entity *User* that seeks to provide an invariant as a domain primitive. This invariant strictly defines the access control roles for a *User* in the salon booking system.

5.2 ADTrees Threat Modeling

Kordy et al. [27] define an ADTree as a node-labelled rooted tree describing the measures an attacker might take to attack a system and the defenses that a defender can employ to protect the system. ADTrees are similar to attack trees since both document potential paths that an attacker can leverage on, to compromise a target system. They both have a tree structure with the goal of the attack as a root node of the tree and the different mechanisms of achieving the attack goal as the leaf nodes. However ADTrees introduce the concept of attack nodes and their antithesis defense nodes as the leaf nodes. Attack nodes in ADTrees reflect an attacker's (sub-)goals while defense nodes reflect countermeasures that neuter or prevent a successful execution of their corresponding attacker's (sub-)goals. ADTrees key contribution in security analysis lies in the provision of refinements of attack paths and definition of a countermeasure for each defined attack path. This means the nodes of an ADTree can be organized in a hierarchical manner with a node having sub-nodes which represent a refinement of a parent node. These refinement correspond to sub-goals of the same type. In some instances, you can have a parent node which does not have corresponding child nodes of the same type. Such a node is called a non-refined node and are used to denote actions that can be viewed as basic in attack tree threat modeling.

For every node of an ADTree, there could be a child of the opposite type that represents a defense mechanism against an attack. Therefore, an attack node may be represented by several children illustrating a refinement of an attack and a single child node which provides a defense mechanism against the attack. The child node offering defense mechanism can also have several children that showcase a refinement of the defense mechanism and a single child of opposite type i.e. an attack node that counters the defense mechanism. While refining a node of an ADTree, one can do a conjuctive refinement where all the child goals of the node under refinement are achieved. To achieve their parent goal, conversely, one can do a disjuctive refinement. With disjunctive refinement of a node, one focusses on achieveing at least one of its child goals. ADTrees core purpose is to model attack and defense scenarios on system resources to be protected. Kordy et al. [27] view the concept of an attack–defense scenario as an interaction between a proponent and opponent.

In their literature, the root of an ADTree represents the proponent's main goal. If the root is defined as an attack node, the proponent in this case will be an attacker and the opponent will be a defender. Conversely, if the root is defined as a defense node, the proponent in this case will be a defender and the opponent an attacker. They propose the following graphical semantics when drawing ADTrees. Attack nodes are depicted by circles and defense nodes by rectangles, as shown in Fig. 5.2. Node relations that represent refinements

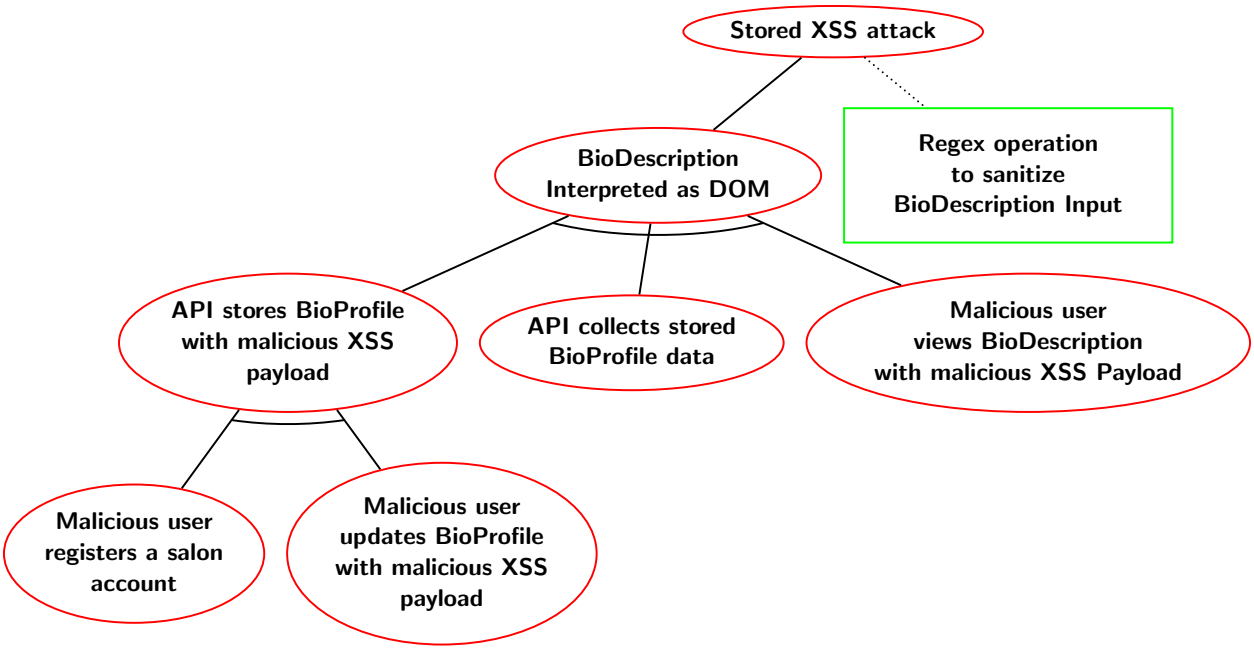

Figure 5.2: Stored XSS ADTree threat modeling

are indicated by solid edges between the parent and child nodes, and countermeasures are represented by dotted edges. To showcase a conjunctive refinement of a node, they use an arc over all edges connecting the node and its children of equal type. To demonstrate an example of an ADTree, we consider a manifestation of stored Cross Site Scripting attack scenario at an API endpoint as shown in Fig. 5.2.

Its root node is an attack, thus the main goal expressed by the tree are scenarios that can lead to execution of a successful stored XSS attack at any given API endpoint and their respective countermeasures. To launch a successful stored XSS attack against a web API endpoint, an attacker can choose multiple vectors of manipulating web API's endpoint accepting user inputs, by providing malicious input which is later interpreted as a Document Object Model (DOM) when loaded on a browser thereby executing the target attack. Assuming we are dealing with an API endpoint of a salon management system for updating a users bio profile as indicated in Fig. 5.2, the following sequence of events need to happen for a successful execution of a stored XSS attack. First, a malicious user A must register an account in the salon management system and must provide the salon web API endpoint for updating his bio profile with a malicious XSS payload. The API will then store the user's bio profile alongside the malicious XSS payload. Any user who will then access user A's bio profile via a web browser will make the uploaded malicious payload stored alongside user A's bio profile to be interpreted as a DOM thereby triggering a successful execution of a stored XSS attack. Therefore, a quick countermeasure against this kind of attacks is to sanitize input at any of the target API endpoints. One way of achieving this is to define a string processing function that strictly

```
...
class BioDescription;

var
bio_description: string
invalid_chars: string
inv
0 < len(bio_description) <= 100

method Init()
  post bio_description = Nil
  invalid_chars = ['<','>', ';', '&', '#', '=','/', '\', '%']
end_method;

method Set_Bio_Description(bio_description: string)
  post bio_description = bio_description
end_method;

method validBio(invalid_chars: string, bio_description: string) is_valid: boolean
/* This method tests for existence of invalid characters in the provided bio_description string */
  pre elems(invalid_chars) <> {}
  post is_valid = true and inter(elems(invalid_chars), elems(bio_description)) = {} or
       is_valid = false and inter(elems(invalid_chars), elems(bio_descriotion)) <> {}
end_method;

end_class;

...

process CreateSalonCustomerProfile(validtoken: token, access_token: token, id: string, user: SalonUser, name: string, phone_number: string,
    bio_description: BioDescription, bio: string) customer_profile: SalonCustomerProfile, error_message: string
ext rd salon_users_table
ext wr salon_customer_profile
pre not exists[i: inset dom(salon_customer_profile)] | i.id = id

explicit
  begin
    if access_token = validtoken and access_token <> nil and elems(access_token) <> {}
    then
      bio_description := new BioDescription
      validbio = bio_description.validBio(bio)
      if validbio = true and user inset(elems(salon_users_table))
        then bio_description.Set_Bio_Description(bio)
          customer_profile = mk_SalonCustomerProfile(id, user, name, phone_number, bio_description)
          salon_customer_profile = override(~salon_customer_profile, {customer_profile-->user})
      else error_message := "Bio description contains invalid characters";
    else error_message := "Http 402, Permission Denied".

  end

end_process
```

Listing 5.1: Sample excerpt of case study API specifications in SOFL

checks for potentially malicious characters that can be interpreted by the browser as DOM objects from our API payload. A sample specification implementation of this counter measure is illustrated in this excerpt listing 5.1 of our case study. The full specifications are availed via this file *salon_api_sofl_implicit.txt* in our github repository [1].

This countermeasure works towards preventing a successful execution of our attack tree root node goal. Compared to other threat modeling techniques, Attack-Defense Trees provide an intuitive graphical representation of different attacks which enable them bridge the gap between stakeholders coming from diverse backgrounds. This enables them to not only detect, analyze, brainstorm, amend results of an attack analysis and document a wide range of attacks but also define reactive countermeasures against the attacks. The

[1] https://github.com/Egalaxykenya/IEICE-journal-paper-emeka/blob/master/SOFL/salon_api_sofl_implicit.txt

stakeholders benefit by relying on a framework that provides a succinct and meaningful structure for a range of potential attack vectors (i.e. methods of gaining unauthorized access to a system) in their system modeling activities.

5.3 Ecore metamodels, models and modeling languages

In Model Driven Engineering (MDE) [64] metamodels and models offer the underlying building blocks. A metamodel defines the abstract representation of a model capturing the concepts of a domain [31]. Metamodels define the frames, rules, theories, and constraints associated with domain concepts [31] and represent models as instances of some more abstract models. A model offers an abstraction of a system or its environment or a combination of both. A model also has a capability of deriving its elements from multiple metamodels rather than being tied to a single metamodel. Therefore, one can define models of reality, and models that describe models i.e. metamodels and recursively models that describe metamodels i.e. meta-metamodels. Cadavid [65], describes a model as a composition of concepts, relationships, and well-formedness. The concepts describe attributes of a domain being modelled, relationships describe the link between the concept and well-formedness describes additional properties that constrain the way domain concepts can be interlocked to form a valid model. Bézivin school of thought [66][67] describes a model as an abstraction of an actual system which can be used to describe the system comprehensively. The intrinsic value of models is pegged on the extent to which a model can help stakeholders take appropriate actions to attain and maintain a system's target goal. Models primary purposes therefore is to reduce a system's complexity through abstraction. MDE promotes this concept by allowing developers focus on the domain problem rather than the technical implementation details when constructing a system's model.

5.3.1 Modeling Languages

Modeling languages are aligned to describe aspects of a system via different sets of symbols and diagrams to minimise the risk of model misinterpretation. MOF is the standard metamodeling language defined by the OMG. However, Ecore, an EMF based metamodeling language is considered a mature standard primarily because it is tailored to Java for implementation and the Eclipse platform which has a huge user base [64]. A modeling language is essentially made up of three parts:

1. An *Abstract syntax* which describes the structure of the language and the way different primitives can be combined together.

2. A Concrete syntax describing specific representations of the modeling language, covering encoding

and/or visual representation. The concrete syntax can be either textual or graphical notations.

3. Semantics which describe the meaning of the elements defined in the language and the meaning of the different ways of combining them.

Modeling languages are broadly categorised into General Purpose modeling Languages (GPML) and Domain Specific Modeling Languages (DSL). DSMLs are tailored for specific domains for example accounting, finance or aviation, with motivation of making it easy for practitioners to describe elements in that particular domain. Examples of DSMLs include SQL for relational databases and HTML for web development. A DSML provides all the elements necessary to construct models in their target domains. These elements are a metamodel, a graphical or textual representation of concepts specified through metamodel and the semantics associated with the domain concepts [65]. On the other hand, GPMLs represent modeling tools that can be applied across multiple domains such as UML and state machines.

A metamodel of a model specifies its structure and meaning. It defines the abstract syntax, the possible elements, and their relations in between. In addition, it specifies its static semantics, the constraints for well-formed models. There are two popular meta-metamodels. The Meta Object Facility (MOF) [68] by the Object Management Group (OMG) which is used as meta-metamodel for the Unified Modeling Language (UML) and Ecore [33] which is part of the Eclipse Modeling Framework (EMF) and based on Essential MOF (EMOF) [68]. We choose Ecore because it has more freely available supporting tool.

5.3.2 Meta Object Facility and Eclipse Modeling Framework

MOF provides a semi-formal approach for defining models and metamodels. MOF was designed to provide a standard metamodeling language for systematic model/metamodel interchange and integration [64]. The Eclipse Modeling Framework (EMF) forms the core technology in Eclipse Development Environment for model-driven engineering. EMF provides a rich set of model-based Model Driven Software Engineering tools offering the following features.

First, EMF enables the definition of metamodels based on the Ecore metamodeling language. Second it provides generator components for manipulating models programmatically, providing API interfaces for constructing models from metamodels. Third, it provides modeling editors to build models in tree-based editors. Morevoer, EMF comes with a powerful API covering different aspects such as serializing and deserializing models to/from XML Metadata Interchange (XMI) as well as powerful reflection techniques. An EMF model is a specification of a data model that can take the form of a UML class diagram or XML schema [31]. Fig. 5.3 gives a hierarchical representation of an*Ecore* metamodel. In Ecore, every object is an

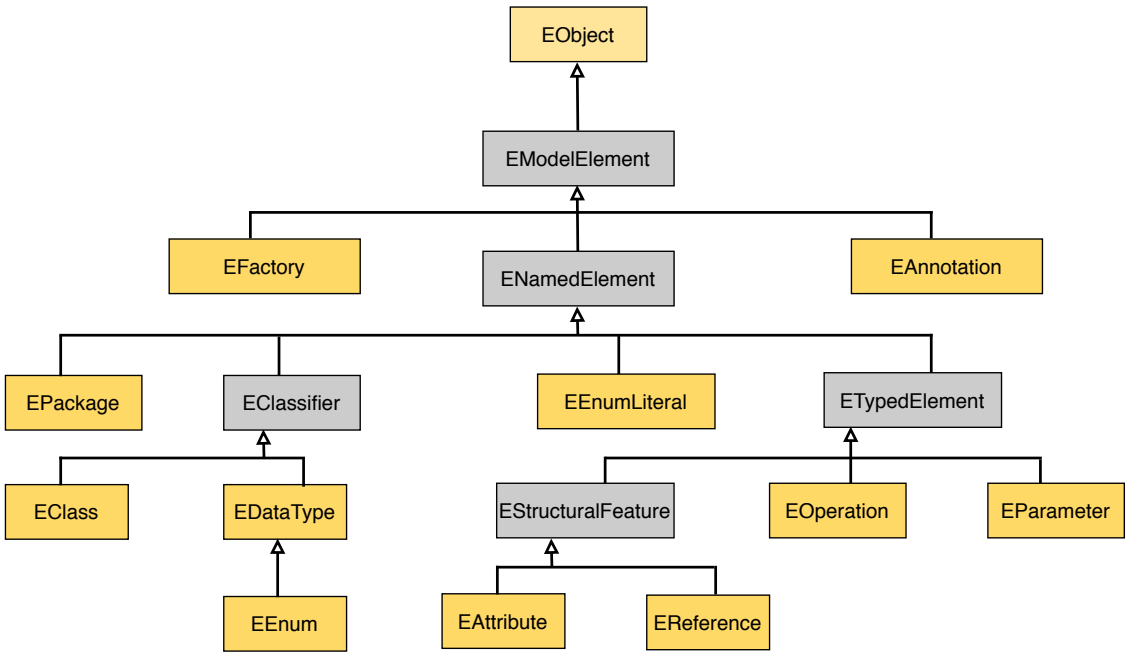

Figure 5.3: Ecore metamodel inheritance hierarchy extracted from [69]

EObject but the root element of an Ecore metamodel is an *EPackage*. An Ecore metamodel consists of the following key concepts:

1. An *Eclass* - Models entities in a domain. An Elass is identifiable by name and usually contains other features such as *EAttributes*, *EReferences* and *EDatatypes*

2. An *EAttribute* models attribute of an EClass. In the hierarchical repsentation of an Ecore metamodel, an EAttribute represents the leaf components of instances of an EClass's data. EAttributes have a name and data type that define their identity.

3. An *EReference* models associations between classes. Associations can be bidirectional with a pairing opposite reference. A containment associations defines a stronger type of referential relation, in which a class contains another class.

4. An *EDatatype* models simple types and acts as a wrapper which denotes primitive or object types.

We use concepts of metamodeling in step 5 of our proposed approach.

5.3.3 Model Transformations

Model transformations are integral, providing mechanisms for querying, synthesizing and transforming models into other models, equivalent formal textual specifications and executable code in Model Driven Engi-

neering. The purpose of models can be wide, spanning from communication between people to executability of the designed software. As such, models and their transformations need to be expressed in a modeling language with some defined notations. Transformation executions are defined based on a set of transformation rules, defined using a specific transformation language. A successful model transformation has to maintain synchronization between source and target models. Model synchronization is often captured by transformation rules. These ensure that a logical correspondence is established between source and target models. Graphical models such as Ecore have been used to capture and encode the concepts of the domain for an application, providing well defined semantics that allow for precise information exchange.

The core objective of modeling is to enable collaborative engagement among the people involved in a project. The involved project members at any given time should be able to access the different parts of the models being constructed by the designers. Naturally, a repository-based approach where designers share a central repository containing the model under construction offers one approach of collaboration. This would mean all the project stakeholders use the same set of modeling tools the designers use to navigate through the models. However, sometimes this approach is not feasible due to overheads such as complexity of modeling tools as most are designed to accommodate the needs of experts therefore training on non-experts is required, as well as need for implementing some access control policies such that that some stakeholders are only granted access to some parts of the models. Moreover, when generating formal specifications such as SOFL from an underlying system model, there need to be consistency and synchronization between the generated SOFL specifications that define the structural representation of the underlying model. This means that if the formal specifications such as SOFL are to be generated manually from an underlying model, there will be difficulties in maintaining consistency and synchronization between the underlying source model and its equivalent SOFL specifications since a change on the underlying model must also be reflected on its equivalent SOFL specifications.

We adopt concepts of model to text transformation in step 6 of our proposed approach. We utilize a generic approach which takes an Ecore metamodel as a source input and yields SOFL formal specifications as target output. The Ecore metamodel enable us create a generic template of a model that is a representation of a RESTful APIs domain model. In particular, we check whether the produced formal specifications define the structure of its source model which conforms to its metamodel and whether all the essential properties are preserved by the transformation. We shall explain this in detail in chapter 5.

5.3.3.1 Epsilon Tool for Ecore Metamodel Construction and Model Management

Epsilon [70], standing for Extensible Platform of Integrated Languages for model management, is a platform for building consistent and interoperable task-specific languages for model management tasks such as model

transformation, code generation, model comparison, merging, refactoring and validation. It offers a set of task-specific languages for different model management operations. For example, EOL (Epsilon Object Language) is used for direct model manipulation, including creating new models, querying, updating, and deleting model elements and forms the core of other languages in the Epsilon suite of languages. Other Epsilon languages include; ETL (Epsilon Transformation Language), for model-to-model transformation, EGL (Epsilon Generation Language), for model-to-text transformation and Epsilon Validation Language (EVL) for model validation.

5.3.3.2 Epsilon Generation Language

EGL is a template-based model-to-text transformation language that is implemented atop the Epsilon model management platform [71] language. EGL can be used to achieve the transformation of models into various types of textual artefacts, including executable code, images, or even formal specifications. EGL provides several features that simplify and support the generation of text from models. These include language-independent merging engine for preserving statically generated text (hand-written) and formatting algorithms which provide mechanisms for achieving traceability that links generated text with their source models. Epsilon offers an abstraction layer called EMC (Epsilon Model Connectivity). EMC specifies an API against which drivers for different modelling technologies are implemented. This enables different model management languages such as Epsilon Verification Language, Epsilon Transformation Language, Epsilon Object Language, etc. manage models captured using different modelling technologies. An EGL program comprises of one or more sections with a mix between static and dynamic sections. The contents of static sections are emitted in verbatim and appear directly in the generated text. The contents of dynamic sections are executed and are used to control the text that is generated. An EGL program comprises of one or more sections with a mix between static and dynamic sections. The contents of static sections are emitted in verbatim and appear directly in the generated text. The contents of dynamic sections are executed and are used to control the text that is generated. EGL re-uses Epsilon Object Language syntax for structuring program control flow, performing model inspection and navigation, and defining custom operations.

EGL also provides syntax for defining sections which are dynamically generated. This provides a convenient mechanism for generating up to date text for the dynamic sections which reflect the changes done on the source model. Similar syntax is often provided by template-based code generators. The tag pair [% %] is used to delimit a dynamic section. Any text not enclosed in such a tag pair is contained in a static section. The code snippet shown in listing 5.2 illustrates the use of dynamic and static sections to form a basic EGL template. Any EOL statement can be contained in the dynamic sections of an EGL template.

```
1 [% for ( i in Sequence{1..5}) { %]
2 i is [%=i%]
3 [%}%]
```

Listing 5.2: Sample EGL template with EOL statements

5.4 REST and REST concepts

The concept of REST was introduced by Roy Fielding in his PhD dissertation, *"Architectural Styles and the Design of Network-based Software Architectures"* [3]. REST relies on HTTP protocol for data communication and revolves around the concept of resources where each component is considered as a resource. These resources are accessed via a common interface using HTTP methods such as GET for retrieving a resource, PUT for updating a resource, POST for creating a resource and DELETE for removing a resource. Contrary to other web services, REST is an architectural style and protocol agnostic. The REST architecture focuses on providing access to a resource for a REST client to access and render it [3]. It utilizes Uniform Resource Identifiers (URIs) in identifying each resource and provides several resource representations such as XML, JSON, Text etc. to represent its type. For an API to be considered RESTful, it needs to satisfy the design characteristics commonly referred to as REST constraints [3] i.e., *Client-server architecture*, *Statelessness*, *Caching*, *Uniform Interface*, *Layered systems*, and/or *Code on Demand* (optional). A detailed description of RESTful web APIs is given in [1].

5.4.1 ResourceType and ResourceIdentifierPatterns

A *ResourceType* represents a RESTful API concept that models an object and its set of properties. It is defined as an abstract *EClass* with a name in our approach. It has an attribute *maxResources* which specifies the number of resources allowed. The uniform interface REST constraint dictates that activities which transcend create, read, update, and delete (CRUD) operations, must be modeled in a different way. For example, if we are creating an Online Salon Booking API with a capability for suggesting the best salon, a suitable workflow needs to be defined: salons can be suggested, and customers must share their reviews. Such a suggestion can be modeled as an *ActivityResourceType*. An *ActivityResourceType* is normally a nominalisation of an activity. A *ResourceIdentifierPattern* describes a URI to a *ResourceType*. Since we are specifically modeling RESTful APIs and by extension adhering to REST semantics, every *ResourceType* must have at least one *ResourceIdentifierPattern*. A *ResourceIdentifierPattern* is abstract and can be a *SimpleIdentifier* which is described using a string or a *ComplexIdentifierPattern* which uses the values of the *ResourceType's* Attributes. A *ResourceType* contains an unordered set of named *ResourceElements*, like attributes and links.

5.4.2 DataTypes and Attributes

Attributes specify a *ResourceType's* properties and conform to a defined *DataType*. A *DataType* can be a *PrimitiveDataType*, a *domain primitive* or a *CollectionType*. *PrimitiveDataTypes* are identified by their name, for example, integers and strings. A *CollectionType* represents an ordered set of values and references the *DataType* of its contained elements.

5.4.3 Method, MethodType and Parameter

REST-based services use HTTP interfaces, such as GET, PUT, POST and DELETE, to maintain uniformity across the web. The uniform interface, enforces a *MethodType*. A *MethodType* is identified by its name, that correlates with existing HTTP verbs. A *ResourceType* is associated with a set of supported Methods and must have a *MethodType*. The Method element defines the behaviour encoded in the API and determines the set of*MediaTypes* consumed or produced by the API. In addition, every Method can define parameters which can be contained in a consumed *MediaType* or in the resource identifier.

5.4.4 Link and RelationType

Links support hypermedia as the engine of application state (HATEOAS) [3]. Each Link can define a media type independent RelationType [72]. This means the client is aware of the relation existing between two resources and which method requests with which meaning can be sent to the target link. A *RelationType* can contain pagination information like next or previous. An *InternalLink* refers to one target *ResourceType*. To model links to resources outside the current application *ExternalLinks* are used. These only go to the extent of defining resource identifiers.

5.5 SOFL

Structured Object-Oriented Formal Language (SOFL), provides a formal but comprehensible language for both requirements and design specifications, and a practical SOFL method for developing software systems. SOFL integrates different notations and techniques on the basis that they are all needed to work together effectively in a coherent manner for specification constructions and verification [15]. The SOFL method offers the following features:

1. SOFL integrates both structured and object-oriented methods in specification construction, by leveraging on their advantages and to avoiding their disadvantages. Structured methods offer a top-down

approach for constructing specifications, that starts from the top level module. The approach then encourages decomposing high level operations defined in the modules into low level modules.

2. It supports an evolutionary three-step approach to developing formal specifications. This evolutionary approach starts with specifications being defined informally, then semi-formally, and ultimately being transformed to formal specification. The informal specification is usually written in a natural language. This provides the foundation for deriving the semi-formal specification in which SOFL syntax is somehow enforced to a certain extent providing a blend of formal and informal parts. The ultimate formal specifications are then derived from the semi-formal specification by formalization of the informal parts.

3. It adopts rigorous review and testing for specification verification and validation. Specification verification aims to detect faults in specifications. Techniques for rigorous review are adopted from the integration of formal proof and fault tree specification analysis methodologies that stem from safety analysis. The reviews must be done on a precise ground, and supported by a rigorous mechanism [73] [74]. Rigorous reviews are usually less formal than formal proofs, and are easy to conduct.

The SOFL specification language offers the following features:

1. It integrates textual and graphical notations. The graphical notation Data Flow Diagrams adopt Petri nets [75] operational semantics and are used to describe comprehensibly the architecture of specifications while the precise definition of the components occurring in the diagrams is achieved by adopting VDM-SL [76], with slight modification. A formalized Data Flow Diagram, resulting from the integration, is called Condition Data Flow Diagram (CDFD).

2. It provides a hierarchical organisation of CDFDs and their associated modules to help reduce complexity and to achieve modularity of specifications.

3. Leverages on *Classes* to model complicated data flows and stores. A store is like a file or database in a computer system. It offers data that can be accessed by processes in a CDFD or by different CDFD in the hierarchy. The value of a store can be used and changed by a SOFL process.

A SOFL formal specification is represented as a group of *modules* organized in a hierarchical manner. Every module include related *processes* that specify the expected functions, the *datastores* which specify the data resources that can be accessed by the process, in addition to the *invariants* that specify the constraints that are to be conformed by the process and data stores. The module offers an encapsulation of the data stores, functions and invariants. A process is composed of five parts *name, input ports, output ports, pre-condition* and *post-condition*.

The input and output ports specify the input and output variables of the process. The pre- and post conditions define the semantics of the process interpreted as follows: When one of the input ports is available, it implies that all of its input variables are bound to specific values of their respective types and the process will be executed. The result of the execution may imply that one of the output ports is made available. This means all of their respective output variables are bound to particular values with specific types. If the pre-condition is satisfied by the input variables before the execution, the output variables are required to satisfy the post-condition after the process execution, provided that the execution terminates. Listing 5.3 shows a typical SOFL formal specification and its corresponding CDFD is shown in Fig. 5.4.

The SOFL formal specifications describe two modules i.e. *Arithmetic* and *Arithmetic_decom*. The module *Arithmetic_decom* is nested within the module *Arithmetic*. Each module contains a number of processes where each process describes an independent system function. Module *Arithmetic* consists of processes *A, B* and *C*. Process *C* is represented as the decomposed module *Arithmetic_decom* in Listing 5.3 lines 17 - 29. These three processes are connected by data flows, which represents the overall function of module *Arithmetic*. The decomposed lower level module *Arithmetic_decom* consisting of processes E and F represent process C in the parent module. An interpretation of the formal specifications of process A is as follows. It consists of one input port and two output ports separated by notation |. It takes x of integer type as the input variable and produces either y or z as the output variable. Its pre-condition is set to true and its post-condition requires that the output variable y is equal to the square of x if x is greater than 0, and the external variable D will be updated by the following condition $D = \tilde{}D + \mathrm{x}$; otherwise variable z will be made available. The $\tilde{}$ sign before the variable D symbolises the initial value of the variable D before it is updated by the process A.

The semantics of SOFL CDFD are interpreted as follows: In each CDFD, a process is represented by a rectangle box with a name in the center. Each input port is denoted by a narrow rectangle on the left part of the process box, which receives input data flows. Each output port is denoted by a narrow rectangle to the right part of the process which produces output data flows. Multiple input and output ports are denoted by multiple narrow rectangles to the left and right parts of the process respectively. The pre- and post conditions are denoted by rectangles located in the upper and lower parts of the process. While using a supporting tool, a mouse click on these areas would give access to the pre- and post conditions respectively. In addition, CDFDs put focus of attention on data flow but not on control. Therefore, there is no explicit linking of input and output ports within a process [77].

More details about the SOFL specifications language can be found from the SOFL book [15].

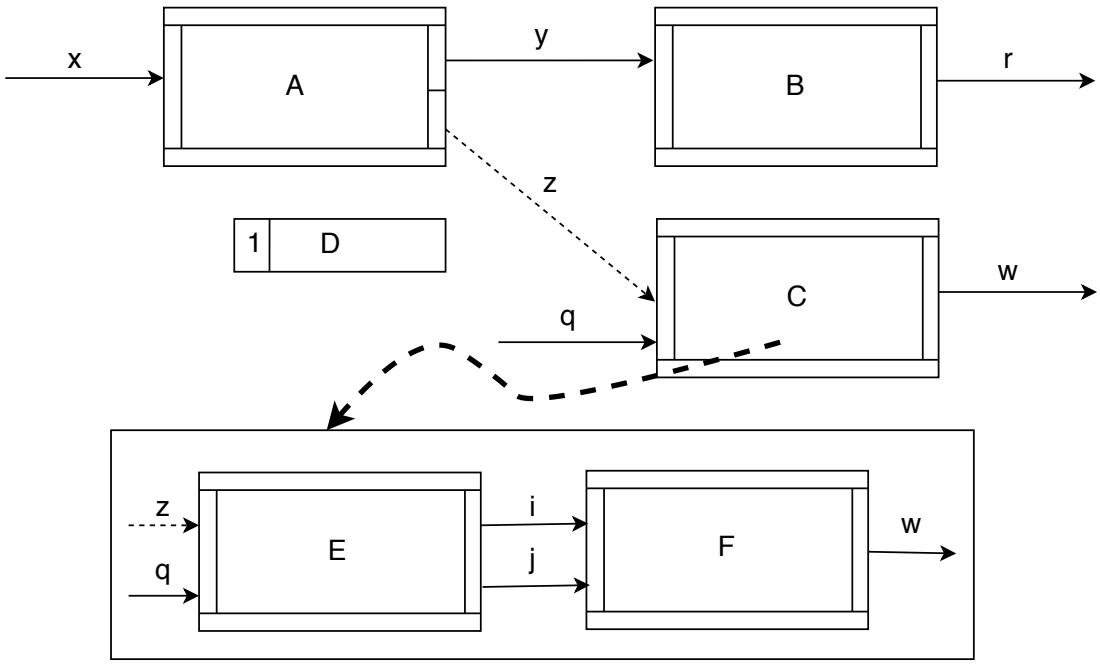

Figure 5.4: An Example of a SOFL CDFD

5.6 Summary

In this chapter, we have described the concepts of the building blocks that we adopt in our proposed approach. These building blocks provide foundational concepts which our proposed approach adopts at some of its steps. SOFL provides the formal engineering specification language for the target API specifications. ADTrees provide a threat modeling approach with provision for not only defining the attack vectors but also defining a mitigation against each of the identified API attack vector. We utilise ADTrees in step 3 of our proposed approach. Domain Driven Design and the concepts of defining API attack mitigation using domain primitives enables our proposed approach achieve the concept of interweaving functional and security API requirements. We utilize DDD concepts in step 4 of our proposed approach. Metamodeling and concepts of Ecore metamodels enable us achieve structural modeling of our target API specifications. We adopt concepts of metamodeling and Ecore metamodels in step 5 and concepts of model transformations in step 6. In the following chapter, we shall describe about our proposed approach, its working mechanisms and the roles played by each of these aforementioned building blocks.

```
1 module Arithmetic
2
3     process A(x: int) y: int | z: sign
4         ext #wr D: int
5
6         pre true
7         post (( x > 0 and D = ~D + x and y = x**2) or ( x <= 0 and bound(z))
8
9     end_process;
10
11    process B(y:int) r: int
12        pre y > 0
13        post r = y * 5
14    end_process;
15
16    /* Process C decomposed into lower level processes E and F */
17    module Arithmetic_decom
18
19        process E(z: sign, q: int) i:int, j: int
20            pre (q > 0 and bound(z))
21            post ( i = q + 1 and j = q ** 2)
22
23        end_process;
24
25        process F(i: int, j: int) w: int
26            pre ( i > 0 and j > 5)
27            post w = i * j
28        end_process;
29    end_module;
30 end_module;
```

Listing 5.3: SOFL formal specification example

Chapter6

The Proposed Approach

In this chapter, we describe our proposed approach in details, highlighting all of its six steps and how we end up achieving the target API specifications expressed in SOFL. Our proposed approach offers a 6 step process (Fig. 6.1) and focuses on interweaving APIs' functional and their respective security requirements. We adopt concepts of Domain Driven Design, threat modeling with ADTrees, Ecore metamodels and SOFL as the building blocks of our methodology. Our approach aims at resolving API security issues both implicitly and explicitly. Implicitly by applying strict invariants on a domain primitive, and explicitly by applying ADTrees to model API threats and vulnerabilities. We rely on official documented repositories such as OWASP API security top 10 [8] and Common Vulnerabilities Exposure (CVE) [78] database as a baseline to guide in identifying common documented RESTful API vulnerabilities. In the following subsequent sections, we describe the 6 steps that define how our proposed approach works.

6.1 Step 1 - Resource Extraction Using RAML Parser

The first step involves generating a flat file with APIs resource listings. In this step, we parse a RESTful API documentation written in RAML with resource definitions as input into an RAML parser [79]. A resource in RAML is identified by its relative Uniform Resource Identifier (URI), which must begin with a slash ("/"). A resource may refer to other resources via steps ("/") in URIs. The resource may be a containment (child) node or otherwise referred to as a nested resource, or non-contained resource. A nested resource will have its URI relative to the parent resource URI. Listing 6.1 shows an excerpt of a sample RAML description of an Online Salon Booking System API which we shall use as a running example. For purposes of brevity we do not showcase the complete listing of the input RAML description file but a full listing of the RAML specifications can be accessed from this github repository [1]. Lines 7-14 show resources externally defined in separate files e.g. API data shapes, resource types and the authenticating security scheme. Lines 16-55 show sample resource type's endpoints and their associated HTTP interfaces with their corresponding request and response data shapes. Parsing the RAML file through an RAML parser yields a flat file (Listing 6.2) with extracted URI resources. This step is fully automated.

[1] https://github.com/Egalaxykenya/IEICE-journal-paper-emeka

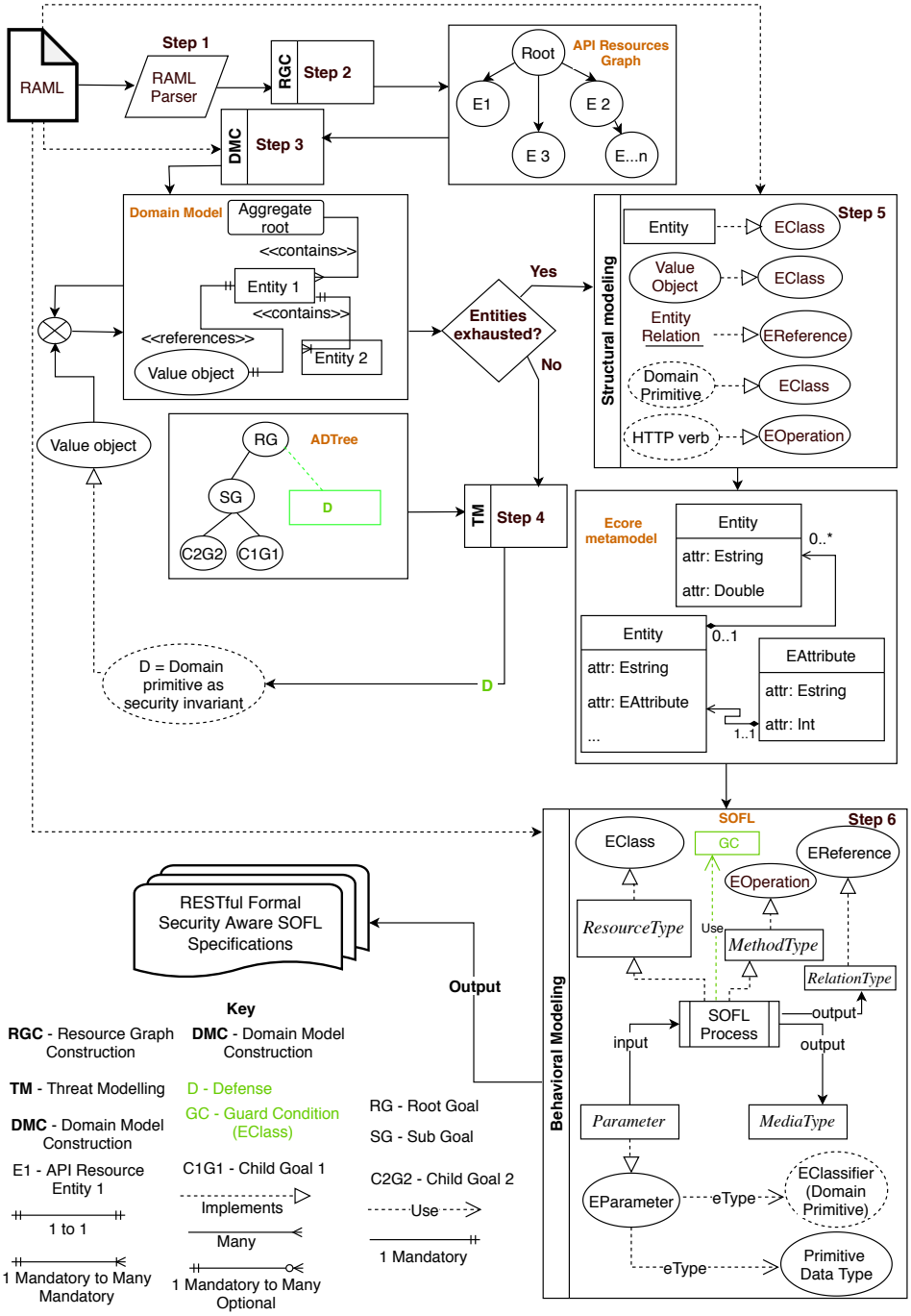

Figure 6.1: Our proposed model. Steps 1 and 2 are automated while the rest of the steps require manual interaction

```
#% RAML 1.0
title: SalonService API
baseUri: http://localhost:8000/api/{version}
version: v1
mediaType: application/json

uses:
shapes: ./dataTypes/shapes.raml

resourceTypes:
collection: !include resourceTypes/collection.raml

securitySchemes:
oauth_2_0: !include securitySchemes/oauth2_0.raml

/salons:
   type:
    collection:
       response-type: shapes.SalonData[]
       request-type: shapes.NewSalonRequestData
   get:
      description: Get a list of Salons based on the salon name
      queryParameters:
        salon_name:
          displayName: Salon Name
          type: string
          description: Salon's name
          example: "Salon Paradise"
          required: true
         ...

    post:
       description: Salon data created correctly for salon business
       body: shapes.NewSalonRequestData
    delete:
     ...
    /{salon_id}:
       type: ...
       get:
        description: Get the salon with `salon_id = {salon_id}`
         responses:
          200:
           body:
            application/json:
             example: |
              {
               "data": {
               "id": "lsVx",
               "name": "Salon Paradise",
               "location": "Chuo Ku,Tokyo",
               "link":"http://localhost:8000/api/v1/salons/SalonParadise"
             },
             "success": true,
             "status": 200
            }
               ...
```

Listing 6.1: Sample RAML description file

```
/salons
/salons/{salon_id}/service-categories
/salons/{salon_id}/service-categories/{category_id}
/salons/{salon_id}/service-categories/{category_id}/salon-services
/salons/{salon_id}/service-categories/{category_id}/salon-services/{service_id}
/salons/{salon_id}/service-categories/{category_id}/salon-services/{service_id}/bookings
/salons/{salon_id}/customers
/salons/{salon_id}/customers/{customer_id}
/salons/{salon_id}/customers/{customer_id}/bookings
/salons/{salon_id}/customers/{customer_id}/bookings/{booking_id}
/salons/{salon_id}/booking-payments
/salons/{salon_id}/customers/{customer_id}/bookings/{booking_id}/booking-payments
/salons/{salon_id}/stylists
/salons/{salon_id}/stylists/{stylist_id}/salon-services
/salons/{salon_id}/stylists/{stylist_id}/bookings
/salons/{salon_id}/users
```

Listing 6.2: Sample flat file with API resource listings

6.2 Step 2 - API Resource Graph Construction

The second step involves automatic construction of an API resource graph that will work as a blue print for creating the target API domain model. The input for this step is the flat file generated from step 1 and the output is a directed graph (digraph) of API resources. We utilize the algorithm defined in Algorithm 1 which takes a list of lists of API resource nodes and the defined API root resource node as an input, and constructs a digraph highlighting all the API resources as an output. For example, the following invocation ENTITYGRAPH*([[salons, service-categories], [salons, salon-services]], salons)* returns *({salons, service-categories, salon-services}, {(salons, service-categories), (salons, salon-services)})*. It is worth noting that in the implementation of the algorithm we had to conduct some pre-processing on the contents of the input flat file such as stripping of ("/") and {id}. This was meant to extract the target API resources from resource URIs which adopt RESTful URIs pattern as depicted by a sample shown in Listing 6.2.

Figure 6.2 shows a sample API entity resource digraph automatically generated by an implementation of the algorithm using Listing 6.2 as a source input. Note the generated digraph abstracts containment and reference relationships between the digraph's resource nodes. A containment relationship describes a relation where a business object represented as a resource entity can contain one or more other business objects with the containing business object known as the parent object while the contained objects are referred to as child objects. A reference relationship describes a relationship between business objects that is not embedding i.e. when you query a business object, its referenced objects are not automatically returned like the case of containment relationships. Similar to step 1, this step is also fully automated.

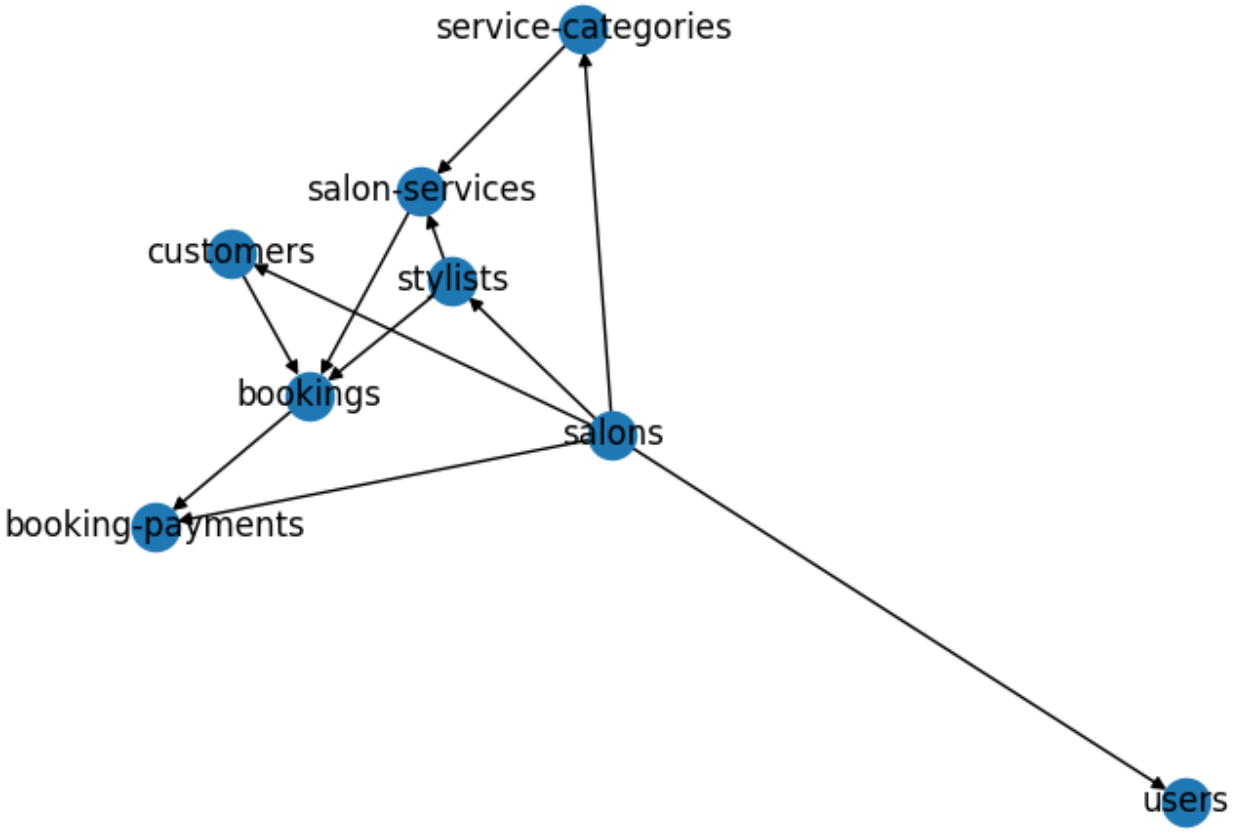

Figure 6.2: API Entity resource Graph

Algorithm 1 API Entity resource graph

```
1:  procedure EntityGraph(L, v):        ▷ L is a list of list of resource names, v is root resource name
2:      let W be an empty set of resource names
3:      let V be an empty set of nodes
4:      let S be an empty set of edges
5:      let M be an empty map from node names to nodes
6:      let r be the root node
7:      for all list l of resource names in L do          ▷ Extract a set of unique resource names
8:          for all resource names n in l do
9:              W ← W ∪ {n}
10:         end for
11:     end for
12:     num ← 1
13:     r ← new Node()                  ▷ Node structure consists of name and order
14:     r.name ← v
15:     r.order ← num
16:     num ← num + 1
17:     M ← M ∪ {v ↦ r}
18:     for all resource names n in (W \ v) do
19:         x ← new Node()
20:         x.name ← n
21:         x.order ← num
22:         num ← num + 1
23:         M ← M ∪ {n ↦ x}
24:         V ← V ∪ {x}
25:     end for
26:     for all list l of resource names in L do          ▷ Extract set of unique edges
27:         for all pairs of adjacent resource names (m,n) in l do
28:             S ← S ∪ {(M[m], M[n])}
29:         end for
30:     end for
31:     return (V, S)
32: end procedure
```

6.3 Step 3 - Domain Model Construction

In step 3, we use the generated digraph as a guide to manually define the API's initial domain model as the target output with an aggregate root corresponding to the root node of the input digraph and the rest of the nodes corresponding to domain model entities. In actual sense, the generated digraph in step 2 is a barebone representation of the target domain model, but what is missing in the domain model at this stage in the distinction between containment and reference relationship between entities. As we construct the domain model, we rely on the encoded business logic defined in the RAML specifications to explicitly define containment and reference relationships between domain model entities. Most importantly, we need to have at the back of our mind that the domain model provides a broad concept that encompasses the resources we choose for the target API and how these resources interact and relate to one another. We define the security

schemes captured in the source RAML specifications that define invariants on related entity attributes as domain primitives in the constructed domain model. This encourages traceability between a domain model and the final generated SOFL specifications in step 6. The importance of an efficient domain layer is key to a successful secure by design API implementation. APIs developed based on the domain layer ensure businesses and security concerns gain equal priority in the view of both business experts and developers. The defined domain model describes the entire ecosystem of the modeled API in the form of Domain Driven Design Concepts of aggregate root, entities, entity relationships and domain primitives. Figure 6.3 shows a sample domain model constructed from an API graph generated in step 2. We adopt UML notations to describe the nature of relationship between any two entity resources or between entity resource and a domain primitive modeled as a value objects.

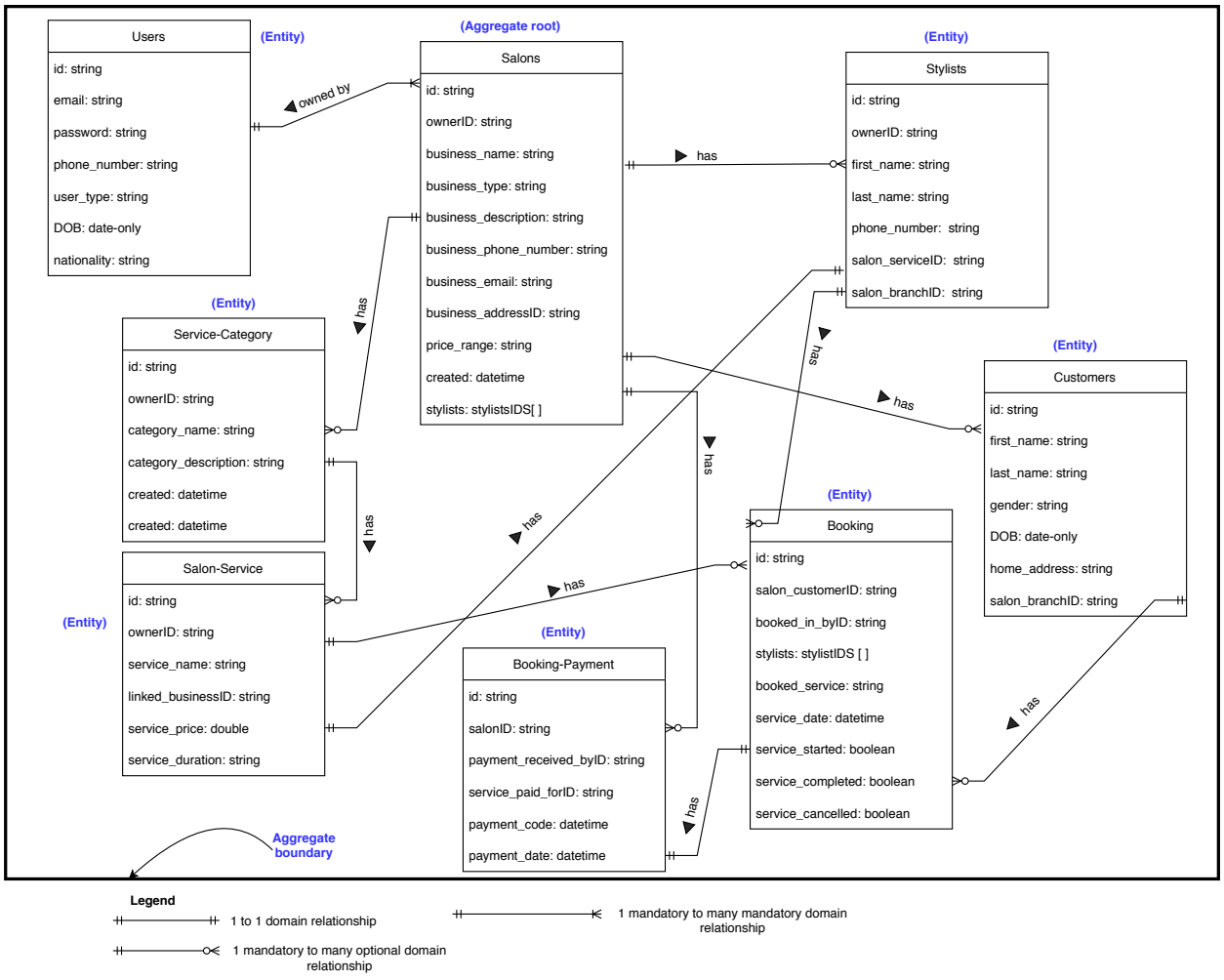

Figure 6.3: Sample domain model created from an API resource graph

6.4 Step 4 - Threat Modeling with ADTrees

The fourth step, which takes our newly defined domain model as input, involves a threat modeling process using ADTrees to identify potential security vulnerabilities in our API domain model and their countermeasures. Countermeasures that can enforce secure constructs on the attributes and behavior of their associated domain entities are modeled as domain primitives. The output of this step is a complete refined domain model with additional security invariants from the threat modeling process defined as domain primitives in the refined domain model. This fourth step achieves our first interweaving of functional and security requirements in an implicit manner. For countermeasures which involve consuming third party services such as rate limiting, request throttling and authentication, we flag them to be modelled as guard conditions and enforce their constraints later in the sixth step during behavioural modeling when defining the pre-post conditions of SOFL processes in the generated SOFL scaffolds.

Likewise, we flag countermeasures to security threats such as Cross-Site Scripting attacks, which can compromise a range of API requests relying on *PUT, POST or PATCH* HTTP methods to expose API resources to their target business functionalities and incorporate them as guard conditions in the SOFL specifications module in step 6.

The threat modelling process involves an analysis of all the domain entities for potential security vulnerabilities. The analysis process is a loop with the condition for proceeding to the next phase based on completion of exhaustive analysis of all the domain model entities representing the API resources. Through the ADTree analysis process, the domain model gets refined courtesy of new value objects being incorporated into the domain model as domain primitives. To illustrate how we achieve this with an example, let's conduct a threat modeling activity on the *Users* resource in Fig 8.2. Our salon Booking API provide end users i.e. salon owners and customers with a web user interface (web UI) where they can register into the salon booking system. As such, our API will provide a users registration endpoint such as *http://salonapi.com/api/salons/users/register* with POST,PUT and PATCH capabilities. The users will be interacting with this endpoint via a form on a web UI. Relying on documented threats on OWASP top ten, to conduct an ADTree analysis on the *Users* resource, we can establish that our endpoint can be susceptible to Reflected Cross Site Scripting attacks (XSS) executed via an inline Javascript. XSS is a common vulnerability affecting web applications in which an attacker can cause a script to execute in the context of another site. To successfully execute an XSS attack against this API endpoint, an attacker can rely on Javascript to submit a malicious input via the form web form that feeds our API endpoint with users registration data. A sample malicious input crafted by attacker relying on XSS exploit is shown in listing 6.3.

```
1 <!DOCTYPE html>
2 <html>
3     <head>
4         <title> Salon Booking System</title>
5             <style>
6                 body {
7                     background: #35363A;
8                     margin: 0;
9              }
10        <style>
11    </head>
12    <body>
13        <form id="register" action="http://salonapi.com/api/salons/users/register"
14            method="post" enctype="text/plain"
15            <input type="hidden" name='{"x":" 'value='","email":"==",
16                "user_type":"&lt;script&gt; alert(&apos; Haha&apos;);&lt;/script&gt;"}'/>
17        </form>
18        <script type="text/javascript">
19            document.getElementById("register").submit();
20        </script>
21    </body>
22 </html>
```

Listing 6.3: Sample malicious input against the salon users registration API endpoint

By default, many HTML forms set their *enctype* atribute to *text/plain*. This instructs the web browser to format the fields in the form as plain text field=value pairs, which an attacker is exploiting to make the output look like a valid JSON (lines 15 - 16). The attacker also includes a javascript script (line 19) to make the form be automatically submitted as soon as its page loads.

For purposes pf brevity and clarity, the attack we are demonstrating here is rather trivial i.e. make an alert pop up window with the words *"Haha!"* appear after the form submission. However, the bottom line is that the same technique can be used for other potentially grievous attacks. It is also worth noting here that as an attacker, we have done some reconnaissance on our target API endpoint and know the structure of the payload it expects.

When the form is submitted, the browser will send a POST request to: *http://salonapi.com/api/salons /users/register* API end point with a *Content-Type* header set to *text/plain* with the hidden form field as the value. Each form element is submitted as name=value pairs. The *<*, *&gt*; and *&apos*; HTML entities are replaced with their literal values <, >, and ' respectively. The name of the hidden input field is *'{"x":"'* and the value is our malicious script. When the two are put together, our API endpoint will see *{"x":"=", "email":"==", "user_type":"<script>alert('Haha!');</script>"* which appears to be like a valid JSON input. We add the extra field "x" to hide the equals sign the browser would have inserted when submitting the name, value pairs. Assuming the client has no client side validation on the email field, our API will do some server side validation on the email field and rejects the email field as invalid echoing back the request payload in the response which will include the malicious script. Since the error response may be served with the default Content-Type of *text/html*, the browser's DOM will interpret the response as HTML and executes the malicious script in the error response payload, resulting in a reflected XSS popup.

Figure 6.4 gives a graphical flow of how this XSS popup attack is executed.

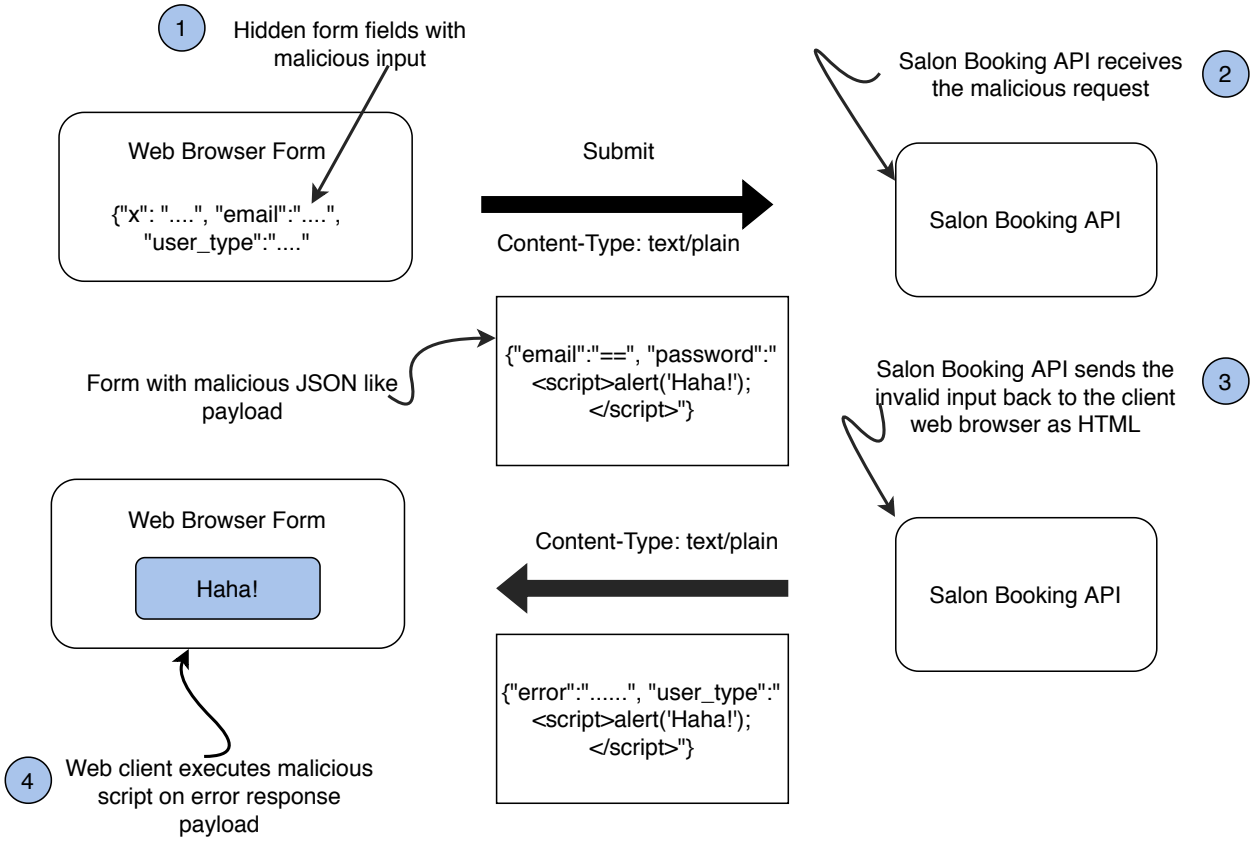

Figure 6.4: Reflected XSS attack flow

Similarly, an ADTree we use for threat modeling activity and define mitigation measures against this type of attack on our API endpoint is illustrated by Fig 6.5. From our ADTree analysis, we define the following mitigation measures which seeks to restrict/validate input from our susceptible API end point.

1. Restrict *Content-Type* header to *application/json* if the the target API consumes JSON payload which is widely used with RESTful APIs.

2. Set security header *X-XSS-Protection* to *"0"*. Browser based inbuilt *X-XSS-protection* filter was introduced in Chrome, Safari and Internet Explorer to stop pages from loading when they suspect instances of reflected XSS attacks. This header can protect users in older browsers that do not support Content Security Policy. However, the header can create security vulnerabilities in otherwise secure websites [80]. It is therefore recommended to use a Content Security Policy (CSP) that disables use of inline Javascript or explicitly turn it off by setting the flag *X-XSS-Protection: 0*.

3. Set *X-Content-Type-Options* to *nosniff* to prevent the browser from guessing the correct *Content-*

Type for our API response. Without this header explicitly set, the browser may ignore our preferred *Content-Type* header and try to make a guess. This can cause a JSON output to be interpreted as HTML or Javascript.

4. Set *X-Frame-Options* to *DENY* to prevent our API responses from being loaded in an iframe. However, modern browsers which support CSP can prevent responses from being loaded in an iframe but its still a worthy defense protection mechanism for old browsers that do not support CSP.

The aforementioned mitigation measures are cross cutting i.e. can apply to any of our API endpoint that services a POST, PUT or PATCH request from users. This therefore qualify as security constructs which we flag them and inject in our specifications as guard conditions of SOFL specifications in step 6. The mode of injection is by including the headers as one of the input variables in the defined SOFL process and have them checked in the pre-post conditions whenever the SOFL process i.e. API request in this matter consumes all of its required input variables.

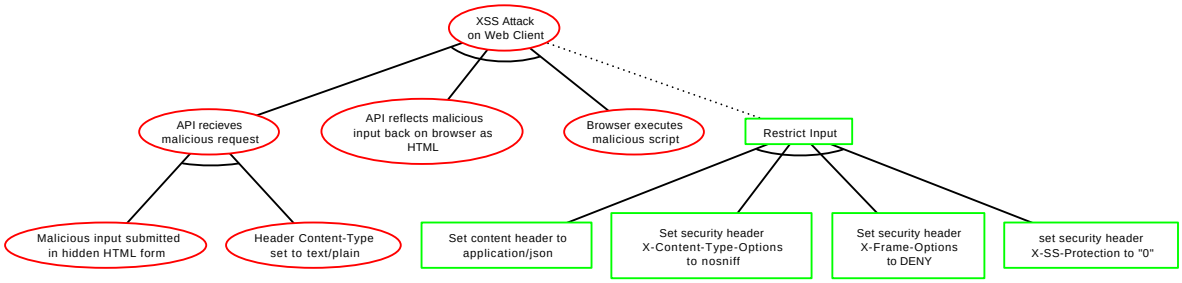

Figure 6.5: ADTree showcasing mitigation measures against Reflected XSS attack

6.5 Step 5 - API Structural Modeling

The fifth step involves creating an Ecore [33] metamodel that describes the structure of our API domain model. In this step, we rely on the refined domain model as input and create an Ecore metamodel that our refined domain model corresponds to, as an output. Specifically, this step encompass structural modeling of our target RESTful API. The structural model describes the possible resource types, their attributes, and relations as well as their interface and representations. We model *entities* as *EClasses*, *value objects* as *EClasses* and their data type as *EDataType*, domain primitives as *EClass* and entity relations as *EReferences*. The modeling of aggregates is omitted since it's a collection of entities and value objects. While modeling a RESTful web API using Ecore metamodels, we adhere to REST semantics i.e. a *ResourceType* must have at least one *ResourceIdentifierPattern*. A *ResourceIdentifierPattern* is abstract and can be a *SimpleIdentifier*

which is described using a string or a *ComplexIdentifierPattern* which uses the values of the *ResourceType's* Attributes. A *ResourceType* contains an unordered set of named *ResourceElements*, like attributes and links. In the following sections, we will briefly explain the REST semantics and their equivalent modeled representations in an Ecore metamodel.

6.5.1 ResourceType

A *ResourceType* models an object and its set of properties. It is defined as an abstract *EClass* with a name. It has an attribute *maxResources* which specifies the number of resources allowed. The uniform interface REST constraint dictates that activities which transcend create, read, update, and delete (CRUD) operations, must be modeled in a different way.

6.5.2 Attributes and DataTypes

We map REST Attributes as *EAttributes* in Ecore. Attributes specify a *ResourceType's* properties and conform to a defined *DataType*. A *DataType* can be a *PrimitiveDataType*, a *domain primitive* or a *CollectionType*. *PrimitiveDataTypes* are identified by their name, for example, integers and strings. *Domain primitives* represents objects consisting of attributes and behaviors that were moved out of a model entity and put into other objects. A *CollectionType* represents an ordered set of values and references the *DataType* of its contained elements. We map REST's *DataType* as *EDataType* in Ecore.

6.5.3 Method, MethodType and Parameter

REST-based services use HTTP interfaces, such as GET, PUT, POST and DELETE, to maintain uniformity across the web. The uniform interface, enforces a *MethodType* mapped as an *EOperation* in Ecore, which is identified by its name, to be defined for all existing methods, i.e., the HTTP verbs. A *ResourceType* mapped as an *EClass* in Ecore, is associated with a set of supported Methods which must have an *EOperation*. The Method element is responsible for the API behavior and determines the set of produced and consumed *MediaTypes*. In addition, every Method can define parameters (*EAttributes*) which can be contained in a consumed *MediaType* or in the resource identifier. A Parameter has a *DataType* mapped as *EDatatype* in Ecore. A Datatype can also be a domain primitive which defines invariants that must be enforced at their point of creation.

6.5.3.1 Link and RelationType

Links are modelled as *EReferences* in Ecore metamodels. In REST semantics, links support hypermedia as the engine of application state (HATEOAS) [3]. Each Link can define a media type independent RelationType [72]. A *RelationType* can contain pagination information like next or previous. An *InternalLink* refers to one target *ResourceType*.

6.6 Step 6 - API Behavioral Modeling and SOFL Formal Specification Generation

The sixth and the final step involves behavioral modeling. The input for this step is an Ecore metamodel from step 5 and the output is formal security aware RESTful API specifications in SOFL language. Our goal here is to define RESTful API behaviors that consist of actions corresponding to their respective HTTP verbs i.e., GET, POST, PUT, DELETE and PATCH. For example, *CreateAction* creates a new resource, an *UpdateAction* provides the capability to change the value of attributes and *ReturnAction* allows for response definition including the Representation and all metadata. To achieve behavioral modelling, we transform our API methods into SOFL processes. A SOFL process definition is by itself a *MethodType* which takes inputs as *Parameters*, yields outputs of either *MediaType* or *RelationType* or both, defines a pre-condition and a post-condition, and can read or write a *ResourceType* to a data store. A *ResourceType* maps to an *EClass* in our Ecore metamodel, *RelationType* maps to an *EReference*, and a *Parameter* maps to an *EParameter* which can have a type that is either a primitive data type, or a domain primitive which defines invariants that must be enforced at their point of creation. For example, an *UpdateAction* can be transformed into a SOFL process complete with pre-post conditions where the inputs are treated as the API's request parameters and outputs as the response including the Representation and all metadata. The semantic constraints of different *MethodTypes* are achieved naturally via the definitions of guard conditions in SOFL's post conditions. This step yields security aware formal RESTful API specifications as an output. While modelling the API behaviors as SOFL processes completed with pre-post conditions and guard conditions [15], our attention to security is drawn to:

- The resources exposed by the API that are to be protected
- Data transfer across the API's trust boundaries and aggregate boundaries
- The security goals that are important such as confidentiality of API resources

- The mechanisms that are available to achieve these goals such as authentication, access control, audit logging and rate limiting

To guide the formalization process of RESTful API behavior in SOFL, we first define the following SOFL formalization techniques with regards to a RESTful API. A RESTful API service D is defined as a set of operations $D = (o_1, o_2, \ldots, o_n)$ where $n \geq 1$. Each operation o_i $(1 \leq i \leq n)$ is represented by an input and output pair of messages in the format $o_i = (inMsg_i, outMsg_i)$. Each input message in $inMsg_i$ is defined as a set of input variables expressed in the format $inMsg_i = \{v_1, \ldots, v_j\}$ $(j \geq 1)$. Similarly, each output message $outMsg_i$ is defined as $outMsg_i = \{v_1, \ldots, v_k\}$ $(k \geq 1)$. The potential functional behaviors are inferred from the resources method definitions in the RAML source file which are encoded as *EOperations* in our API structural model. RESTful services represent business processes which may be organized in a hierarchical manner to represent business goals. Therefore each function can be decomposed further into low-level business processes. The formal representation of a REST service process in SOFL therefore involves 2 steps:

- Modularize REST service associated functions into proper SOFL processes. We adopt the following two rules during the modularization process:
  - Rule 1: If a function $F = \{f_1, ..., f_m\}$ represents a service S, we define a process P_i for each sub-function f_i $(i = 1, ..., m)$ of F.
  - Rule 2: If a function F interacts with a data item x, then we construct a data store d to represent x. A datastore d specifies the expected data resource which is accessed by function F. It represents a necessary RESTful API resource that is shared by several processes.
- Fully formalize the pre- and post-conditions of these processes to precisely express the expected operational semantics upon their associated services.

We formaly define a SOFL process as a five-tuple: $(P, InPortSet, OutPortSet, preP, postP)$

- P is the process name
- $InPortSet = \{inPort_1, inPort_2, \ldots, inPort_f\}$ is the set of input ports of P where $inPort_i$ $(i = 1, \ldots, f)$ define an input port. Furthermore, each input port is expressed as $inPort_i = \{v_{j_1}, \ldots, v_{j_{r_i}}\}$ where $v_k (k \in \{j_1, \ldots, j_{r_i}\})$ is a variable of this port.
- $OutPortSet = \{outPort_1, outPort_2, \ldots, outPort_g\}$ represents the set of output ports of the process P where $outPort_i$ $(i = 1, \ldots, g)$ is an output port. For each output port defined as $outPort_i = \{v_{l_1}, \ldots, v_{l_{s_i}}\}$ where v_k $(k = l_1, \ldots, l_{s_i})$ defines a variable of this port.

- *preP* is the pre-condition of P, which specifies the condition that the input variables need to satisfy.
- *postP* is the post-condition of P, which specifies the condition that the output variables are required to satisfy.

The semantics of a process P with respect to input and output ports corresponds to the interpretation of a CDFD diagram as described in chapter 5, section 5.5 i.e. when one of the input ports in *InPortSet*, say $inPort_i$, is available, then it implies that all of its input variables are bound to specific values of their respective types and that the process P will be executed. If as a result of the execution, one of the output ports in *OutPortSet*, say $outPort_j$, is made available, then it means all of its output variables are bound to specific values of their respective types. If the input variables satisfy the pre-condition *preP* before the execution of P, the output variables are required to satisfy the post-condition *postP* after the execution of the process P, as long as the execution terminates.

To interweave security requirements with functional requirements at this stage, we introduce the concept of SOFL process functional scenarios [81]. The pre- and post-conditions of a SOFL process can be transformed into a number of independent relations called functional scenarios. Let the post-condition $P_{\text{post}} \equiv (C_1 \wedge D_1) \vee (C_2 \wedge D_2) \vee \ldots \vee (C_n \wedge D_n)$, where each C_i $(i = 1, \ldots, n)$ is a predicate called guard condition that contains neither output variables nor output external variables of the SOFL process and D_i is a predicate called defining condition that contains at least one output variable but does not contain any guard condition as its constituent expression [82]. Then each $\tilde{P}_{\text{pre}} \wedge C_i \wedge D_i$ is called a functional scenario, where $\tilde{F}$ for logical formula F of the input/output variables of a process denotes the value of F before starting execution of the process. The pre- and post conditions of a process P can then be transformed into a functional scenario of the form $\equiv (\tilde{P}_{\text{pre}} \wedge C_1 \wedge D_1) \vee \ldots \vee (\tilde{P}_{\text{pre}} \wedge C_n \wedge D_n)$. Each functional scenario $(\tilde{P}_{\text{pre}} \wedge C_i \wedge D_i)$ independently defines how the output of P is defined using D_i under the condition $\tilde{P}_{\text{pre}} \wedge C_i$. Guard conditions enforce invariants that constrain the behavior of their associated processes. In our case we define guard conditions that enforce security constraints thereby achieving the second interweaving of API's functional and security requirements.

6.6.1 Semi-Automatic Generation of SOFL via Model to Text Transformations

To semi-automatically generate the formal SOFL specifications that represent the structure of an API from its underlying model that conforms to a metamodel, we conduct model to text transformations leveraging on the Epsilon Generation Language [71]. EGL is a model-to-text transformation language that is template driven. EGL was natively designed to support code generation but can be extended to generate any form of text file including formal specifications. However, it will be critical to note that our proposed approach is

agnostic to any model-to-text transformation language. In principle, the approach can be implemented using any other model-to-text transformation language, such as Xpand [64], MOFScript[64] and Acceleo [83].

In the generation of SOFL specifications, we generate SOFL signatures. These signatures offer a lightweight mechanism that determine the sections of the program text that must be re-executed in response to a change in its input source model. The generated signature elements that define a module name, a SOFL class name, its variables, methods and method inputs and their types as well as outputs and their types will be dynamically generated whereas signature elements such us the *pre-* and *post* conditions of the SOFL methods as well as its class methods, will be generated statically. The dynamically generated sections are executable sections which are evaluated with respect to the source model while the statically generated sections are written in verbatim, and remain the same during the model to text transformation. when a transformation is first executed, the epsilon tool generates the SOFL signatures and write them to a non-volatile storage. When a transformation is re-executed in response to changes in the source model, the SOFL signatures are recomputed and compared to those from the previous execution. In its dynamic sections, EGL re-uses Epsilon Object Language syntax for structuring program control flow, performing model inspection and navigation, and defining custom operations.

A typical model-to-text transformation is achieved using a module that comprises of one or more templates. A template is a document file with a set of parameters, which specify the data on which the template must be executed, and a set of expressions that define the behaviour of the template. Model to text transformation languages also provide *TemplateInvocations*, which are used to invoke other templates and *FileBlocks* which comprise of rules that can be evaluated for redirecting generated text to a file. The model to text transformation process is achieved via a transformation engine that takes an input as source model and outputs text. The transformation engine creates a *TemplateInvocation* object from an initial template which consist of parameter values and template expressions. The execution of a *TemplateInvocation* involves evaluation of the expressions of its template according to its parameter values. The *FileBlocks* are then evaluated by writing to disk the text generated based on the rules contained within the *FileBlock*.

An overview of our proposed model to text generation technique adopts the same aforementioned principles. We first have to create an Ecore metamodel and a sample model that conforms to it. We use the Epsilon tool [84] to create our Ecore metamodel that gives a structural representation of our API using the Emfatic textual syntax. We then generate a proper XMI-based Ecore metamodel from the Emfatic [70] textual representation yielding a *.core* file. This generation process is fully handled by the Epsilon tool. Next, we register the generated *.core* as an *EPackage*. We then register the generated *.core* as an *EPackage*. This enables us create a model that conforms to our Ecore metamodel. The Epsilon environment provides a wizard for achieving this as we shall illustrate in the case study section. With a model created, conforming

to our Ecore metamodel, we then create an EGL template using our model as its source configuration and its conforming Ecore metamodel by writing EOL scripts that read our model to extract the dynamic parts, which are expressed in EOL syntax alongside the static SOFL signatures. We then create a *.egx FileBlock* with rules which specify the output file bearing the SOFL formal specification, the source EGL template and the location where the output file will be stored.

6.6.2 Preserving Statically Generated Text and Handling of Changes in the Source Model

In this section, we discuss how we handle the preservation the statically generated SOFL specification text in our model to text transformation as well as how changes in the source model are propagated to the generated SOFL text. In real world projects, the source model would evolve over time either to incorporate new requirements or to reflect a refinement of the previously defined requirements for the source model. To handle this, the Epsilon tool provides a mechanism for source incremental[85] model to text transformation. This is achieved through identification of the subset of *TemplateInvocations* that need to be evaluated to propagate the changes from the source model to the generated SOFL text. To preserve the statically generated text, the Epsilon transformation engine evaluates only the dynamic sections of a *TemplateInvocation* ignoring any static sections.

Fig 6.6 describes the mechanism employed by the Epsilon model to text transformation engine for preserving static sections of the generated text as well as achieving source incremental model to text transformation. At initialisation, the transformation engine loads an EGL template as well as its configurations into an EGL template store which prepares access to the target source model referenced by the template. The transformation engine then loads the source model (step 3) into an EOL scripts evaluator. The EOL script evaluator reads into the provided source model and generate SOFL signatures as per the defined project's EOL scripts. The *TemplateInvocation* is then called by the project's *FileBlock* which writes the target SOFL formal specifications into a file for persistent storage. whenever the source module changes, the EOL script evaluator returns a Boolean value *hasChanged* which indicates whether or not the source model differs from the one currently referenced in the EGL template store. If the source model has changed i.e. updated, a *TemplateInvocation* is executed by evaluating the dynamic sections of the template while the static sections are output in verbatim. This has the effect for preserving the static sections of the generated text. When the transformation is complete, the transformation engine informs the EOL scripts evaluator so that it can persist the generated text in a storage.

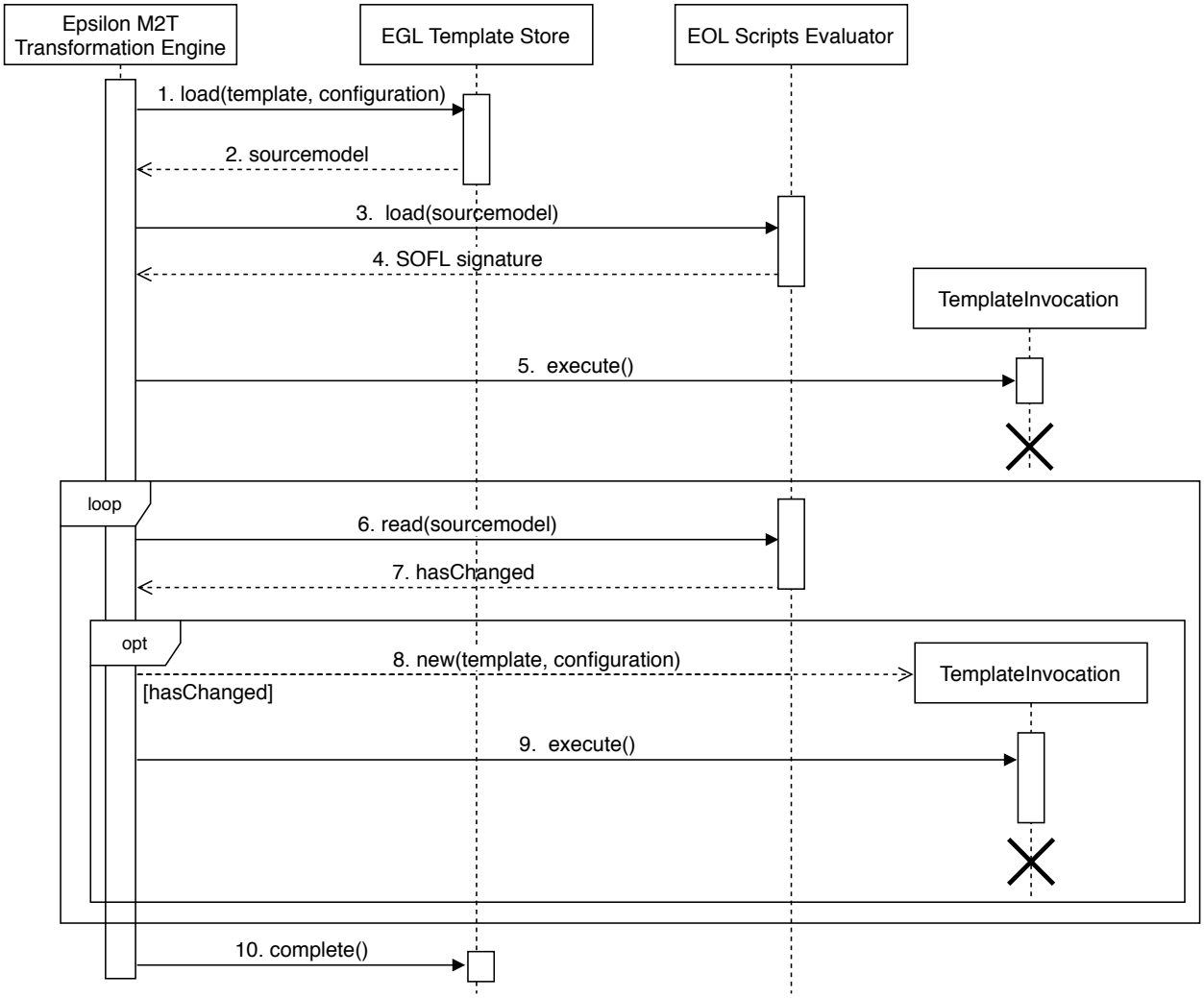

Figure 6.6: A UML sequence diagram describing how the SOFL signatures are generated and how the static sections of the signatures are preserved

```
@namespace(uri="salonbooking", prefix="")
package salonbooking;

class Salon {
    attr String salon_name;
    attr String salon_address;
    val Service[*] services;
    val Stylist[*] stylists;
    val Customer[*] customers;
    val Booking[*] bookings;
}
class Service {
    attr String service_name;
    attr Double price;
    attr ServiceCategory category;
}
class Stylist {
    attr String stylist_name;
    ref Service[*] services;
}
class Customer {
    attr String customer_name;
    ref Booking[*] bookings;
}
class Booking {
    attr int booking_id;
    ref Customer[1] customers;
    ref Stylist[1] stylists;
    ref Service[1] services;
    ref Salon[1] salon;
}

enum ServiceCategory {
    HairCut;
    hairStyling;
}
```

Listing 6.4: Sample Ecore metamodel in EMF text

6.6.2.1 Model to Text Transformation Running Example

In this section, we present a trivial example that demonstrates the proposed approach for generating SOFL specification that yield a structural representation of a model. The example illustrates a simple salon booking system. The key focus of this example is to highlight the steps that lead to generation of the target textual SOFL specifications from a source model that conforms to the salon booking metamodel. We begin by creating an Ecore meta model of the salon booking system using Emfatic textual syntax as shown below.

Epsilon Tool also gives us a provision of generating *.core* file from our Ecore metamodel in Emfatic syntax as well as its diagramatic representation as shown in Fig 6.7.

We register the generated *.core* file as an *EPackage* so as to create a model that conforms to our Ecore metamodel. Epsilon tool provides an interface for creating a model from a metamodel and validating the created model. This model is just a dummy representation which can be used as a bridge for enabling other project stakeholder communicating with other project stakeholders about changes on requirements represented by a domain model. The process is semi-automatic with Epsilon managing relations between model entities as well as validating the datatypes of model entity attributes in conformance with the model's metamodel as indicated in Fig. 6.8.

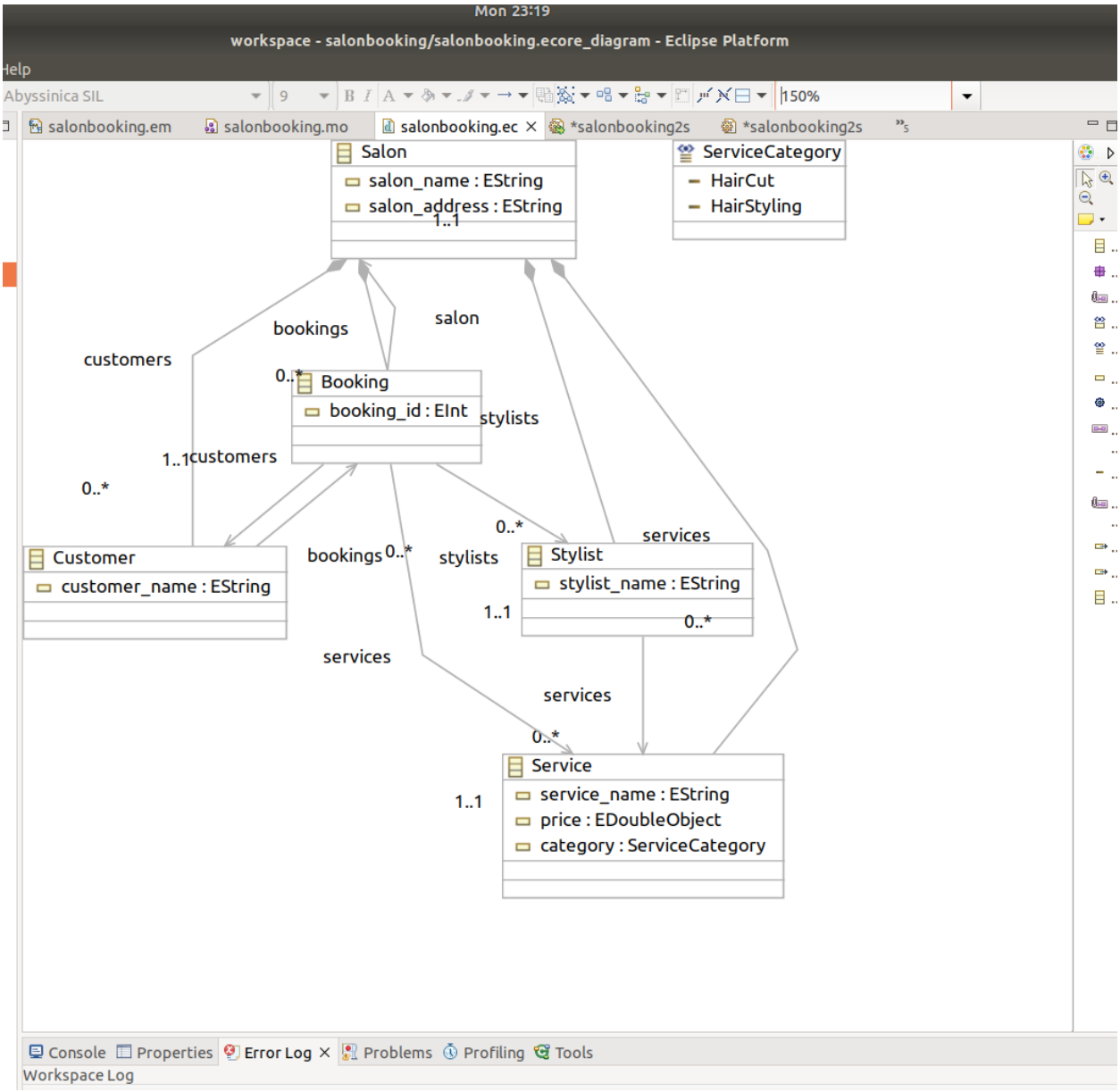

Figure 6.7: A Diagrammatic representation of Ecore meta model generated automatically by Epsilon Tool from Source Ecore meta model in Emfatic textual syntax

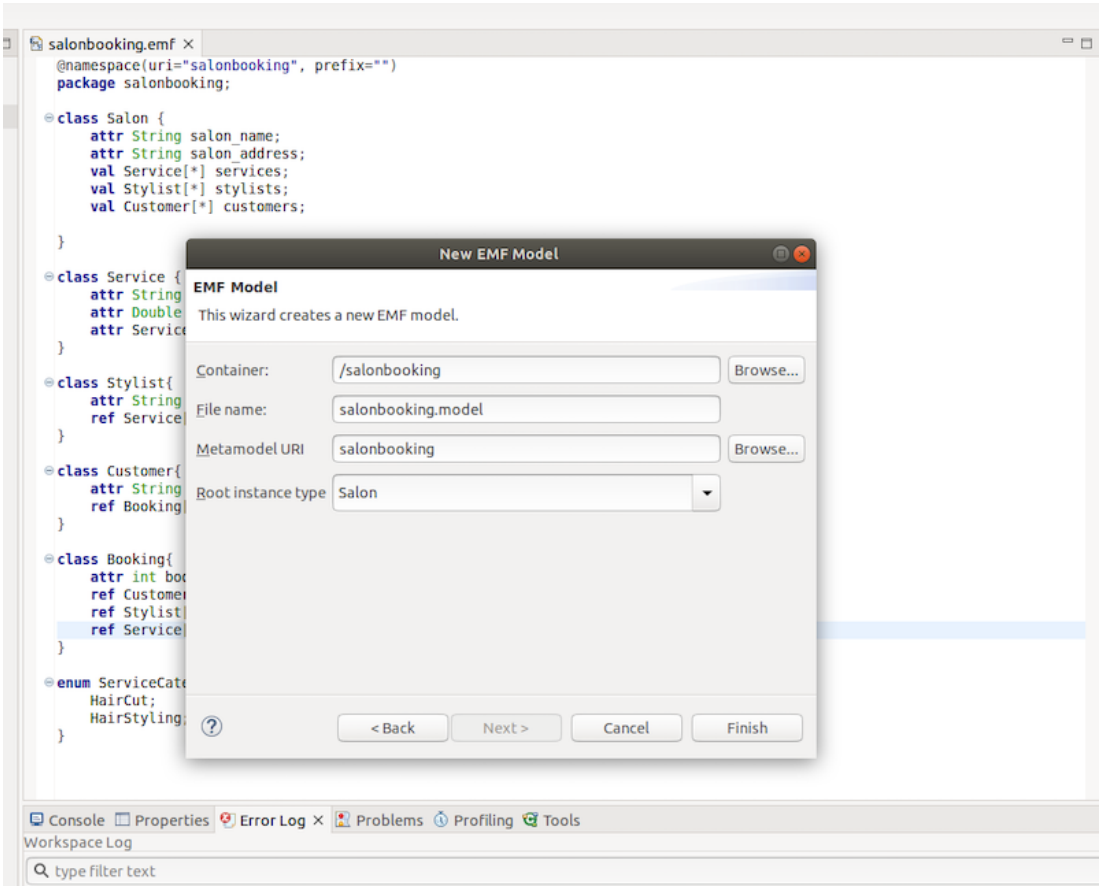

Figure 6.8: Epsilon wizard for semi-automatic generation of source model conforming to a source Ecore meta model

```
pre {
    var package = Salon.all;
}

rule Salon2SoflClass

    transform s : Salon{
        template : "salonbooking2sofl.egl"
        parameters : Map {"package" = package}
        target : "home/emeka/m2t" + package + s.salon_name + ".txt"
    }
```

Listing 6.5: Sample .egx file with single transformation rule

With the source model that conforms to a meta model available, the next step involves generating two types of files, i.e *.egx* and *.egl*. The *.egx* file defines the rules that generate the ultimate textual file with SOFL specifications artifacts. It specifies parameters such as the source *.egl* file from which it reads from and the target physical location in which the generated textual file will be stored. The code snippet below shows a sample *.egx* file with a single transformation rule indicating the source *.egl* file and the target storage location of the to be generated textual file with a dynamic naming scheme.

The *.egl* template file is what yields the power behind the model to text transformation process. It offers a template language tailored for model-to-text transformation (M2T). We write the static sections in verbatim and define the dynamic sections by delimiting them using [% %] tags. The dynamic sections re-uses EOL's mechanism for structuring program control flow, querying the model, inspecting the model, navigating the model and defining custom operations on the model. In general, the *.egl* file provides the syntax for defining the dynamic and static output sections, which provides a convenient way for yielding textual output from within both dynamic and static sections. A sample *.egl* file that generates a simple SOFL class signature is shown in the following code excerpt in listing 6.6.

The above EGL template should yield the following SOFL class signature as a generated text file as shown in listing 6.7.

6.7 Summary

In this chapter, we have discussed about our proposed model driven approach and explained detail the six steps that our approach encompass. In addition, we have discussed how we can model the encoded behaviours of a RESTful Web API as well as describe the techniques for semi-automatically generating API's SOFL formal specifications via model to text transformations. Considering model to text transformation has been around for some time, positioning itself as one of the most trans-formative technologies in model driven engineering, the need for maintaining synchronization between source and target model with support for formal specification as the transformation target is becoming paramount. The model to text transformation

```
[%=var salonclasses : Sequence %];
[%=salonclasses = Salon.all %];
[% for (salonclass in salonclasses){%]
 [% if ( salonclass.Class == "Salon") {%]
class [%=salonclass.Class%];

type
 [%=salonclass.salon_name%]:[%=salonclass.salon_name.type%];

var
SalonStore: seq of [%=salonclass.salon_name%];

method Init()

end_method;

method CreateSalon([%=salonclass.salon_name%]:[%=salonclass.salon_name.type%])

ext wr Salons:SalonStore

pre

post

end_method;
end_class;
    [%}%]
[%}%]
```

Listing 6.6: Sample .egl that generates a simple SOFL class signature

technique discussed here enriches our proposed approach by encouraging synchronization between a model and its equivalent SOFL specifications. This enables any structural change made on a model be reflected on its equivalent SOFL representation via an automated process without the need of a specialized tool to achieve the same. We hope our proposed approach will inspire related works focusing on generating formal specifications that conforms to the grammar of an underlying system model. In the next chapter, we shall discuss about specification testing, a formal engineering methodology for testing the validity of SOFL specification generated via our proposed approach that represents a given RESTful API.

```
1  class Salon;
2
3  type
4  salon_name : String;
5
6  var
7  SalonStore: seq of salon_name;
8
9  method Init()
10
11 end_method;
12
13 method CreateSalon(salon_name: String)
14
15 ext wr Salons:SalonStore
16
17 pre
18
19 post
20
21 end_method;
22 end_class;
```

Listing 6.7: Generated SOFL class signature

Chapter7

Formal Specification Testing

7.1 Introduction

Our proposed approach yields final formal API specifications expressed as a set of system modules encapsulating the functions, data resources and constraints. In particular, the API's request methods are specified as SOFL processes. The input and output data structures of these processes are formally defined while the pre- and post-conditions maybe expressed in formal or informal language.

7.2 Formal Specification Testing Technique

To verify whether the interweaved functional and security requirements implement all expected functions correctly and satisfy the desired security constraints, we can optionally perform specification testing to verify whether the specifications reflect the user requirements. Given a process $P \equiv (\tilde{P}_{\text{pre}} \wedge C_1 \wedge D_1) \vee \ldots \vee (\tilde{P}_{\text{pre}} \wedge C_n \wedge D_n)$ where $n \geq 1$, if we define a test set T, then T is said to satisfy the scenario-coverage of P if and only if $\forall_{i \in \{1,\ldots,n\}} \exists_{t \in T} \cdot P_{\text{pre}}(t) \wedge C_i(t)$. We interpret this as: A test set T satisfies the scenario coverage for the process P if and only if for any functional scenario, there exists a test case in T such that it satisfies the conjunction of the pre-condition $\tilde{P}_{\text{pre}}$ and the guard condition C_i. The test set T ensures that every functional scenario with its associated security constraint is covered appropriately. To check for conformance of a process P specifications relative to user requirements of an API service operation o, we generate a test case t for each functional scenario $f_i \equiv \tilde{P}_{\text{pre}} \wedge C_i \wedge D_i$ using concrete input values and analyze the test results in order to determine whether violations of security constraints are detected. If r is the result of an API service operation o indicated by user specification using a test case t, and r' is the animation [86](explained shortly) result of a process specification P using a test case t, if r' of the process P matches r, then we can confirm that process P property of operation o represents the users requirements and its associated constraints.

Implicit SOFL specifications do not indicate algorithms for implementations. However, they are expressed with predicate expressions involving pre and post conditions for a process and can be evaluated if all variables involved are substituted with concrete values of their types with results of such evaluations being truth values *true* or *false* [15]. For our specification testing, we further apply process animation technique to obtain the set of concrete values of output variables for each functional scenario. An analysis of a test

```
module Salon_API;
...
token = string;
headers = composed of

SalonData = composed of
              id = string
              owner = SalonUser
              business_name = string
              business_type = string
              business_description = string
              business_phone_number = string
              business_email = string
              business_address = AddressData
              price_range = string
              created = Timestamp
            end;
SalonTable = seq of SalonData;
var
salons_table: SalonTable

inv
forall[i,j: inds(salons_table)] | i <> j
            => salons_table(i).id <> salons_table(j).id;

process AddSalon(validtoken:token, access_token:token, salon: SalonData)
                response_message: string
    ext wr salons_table
    /* Pre condition: id of new salon must be unique */
    pre not exists[i:inds(salon_table)] | salons_table(i).id = salon.id
    post access_token = validtoken
        and salons_table = conc(~salons_table, [salon])
        and response_message = "HTTP 200"
        or salons_table = ~salons_table and elems(access_token) = {}
        and access_token <> validtoken
        and response_message = "HTTP 401"
        or salons_table = ~salons_table and access_token = Nil
        and access_token <> validtoken
        and response_message = "HTTP 401"

end_process;
...
```

Listing 7.1: SOFL formal specification for RESTful API AddSalon

results is done by comparing evaluation results with the analysis criteria. The analysis criteria is a predicate expression representing the properties to be verified. If the evaluation results are consistent with the predicate expression, the analysis show consistency between the process specification and its associated requirement. We generate the test cases for both input and output variables based on the user requirements.

7.3 Running Example

A simple running example can be used to demonstrate how we conduct specification testing to test if the specifications meet its critical requirements and also provides the desired functionality. In our case study, a RESTful API request operation *AddSalon* for creating a salon object needs to be checked on whether it satisfies its interweaved security requirement that requires an access token i.e. *guard condition* to be provided for a successful authorization for creation of a salon object. The required function on this operation is formally given as indicated in Listing 7.1. The process *AddSalon* takes a *validtoken*, an *access_token* and *salon* as

input variables and returns an appropriate HTTP *response_message* as an output variable. The *validtoken* is a string constant used to verify the validity of the provided *access_token* by returning a boolean value. If we examine the pre-post conditions of the process *AddSalon*, we get three functional scenarios as follows:

(1) *access_token=validtoken* ***and*** *salons_table = conc(˜salons_table, [salon])* ***and*** *response_message = "HTTP 200"*

(2) *access_token <> validtoken* **and** *elems(access_token) = {}* ***and*** *salons_table = ˜salons_table* ***and*** *response_message = "HTTP 401"*

(3) *access_token <> validtoken* **and** *access_token = Nil* ***and*** *salons_table = ˜salons_table* ***and*** *response_message = "HTTP 401"*

We can generate test data from each functional scenario through specification animation as earlier described. Table 7.1 shows sample test cases covering the three functional scenarios and their corresponding results. The test cases generated are usually based on test targets which are predicate expressions, such as the pre and post conditions of a process. To cover for functional scenario (1), we provided a test case *(validtoken, "xvfKjT", salon)* for the required input variables *validtoken*, *access_token* and *salon* object respectively. After executing the test case on a process specification *AddSalon*, the output value of variable *response_message* is equal to the expected output value inferred from the defining condition *response_message = "HTTP 200"*. For functional scenario (2), we run the test case *(validtoken, " ", salon)* and the value of the output variable *response_message* corresponds to the expected results. Running the test case *(validtoken, Nil, salon)* for functional scenario (3) also yields a value for the output variable that corresponds to the expected results as per interpretation of the user requirements. Therefore, we can determine that the process specification *AddSalon* does satisfy its critical requirements and as per its user requirements. Since our focus is on the relationship between input and output variables, and security concern rather than function, to simplify our presentation, the current test does not inspect data stores. For example, if empty access token is given to *AddSalon*, then *salon_datastore* should be untouched. In addition, the *validtoken* which could be provided by a third party service such as an authenticating service, is assumed to be always valid. However, these specifications are not explicitly captured in the test. We could incorporate data stores by extending our test cases. It is also worth noting that when testing for conformance of a process specification to its associated service operation, we only need to observe the execution results of the process by providing concrete input values to all of its functional scenarios analyzing their defining conditions relative to user requirements.

Functional Scenario	**Test Case**	**Execution Result**	**Expected Result**
access_token = validtoken ***and*** *salons_table = conc(˜salons_table, salon)* ***and*** *response_message = "HTTP 200"*	(validtoken, "xvfKJgT", salon)	"HTTP 200"	"HTTP 200"
access_token <> validtoken **and** *elems(access_token) = {}* ***and*** *salons_table = ˜salons_table* ***and*** *response_message = "HTTP 401"*	(validtoken, " ", salon)	"HTTP 401"	"HTTP 401"
access_token <> validtoken ***and*** *access_token = Nil* ***and*** *salons_table = ˜salons_table* ***and*** *response_message = "HTTP 401"*	(validtoken, Nil, salon)	"HTTP 401"	"HTTP 401"

Table 7.1: Testing RESTful service operation AddSalon.

7.4 Summary

In this chapter, we have discussed the techniques for Specification Based Testing of RESTful web APIs SOFL specifications generated using our proposed approach. Specifically, the technique for testing stateless RESTful API endpoints expressed as SOFL processes are described. The construction of the functional scenarios for each API endpoint is illustrated. In the next chapter, we shall discuss in details a case study we conducted to demonstrate the practicability of utilizing our proposed approach on real projects.

Chapter8

Case Study and Experiments

In this chapter, we present the case study and experiments, and discuss some of the challenges we uncovered. The objective of the case study was to evaluate the effectiveness of our proposed framework in designing a security aware RESTful web API with the goal of achieving the interweaving of functional and their related security requirements.

8.1 Case Study Set Up and Experiments

To conduct our case study, we needed to set up the necessary software environment for Salon Booking System and its RESTful web APIs as well as supporting tools. The following sections describe the environment, the set up tools as well as a detailed description of the case study implementation.

8.1.1 Set up Environment and Software Tools for the Case Study

The specific software environment and tools are listed in Table 8.1 Django RESTframework is a python

Software/Environment	**Main Function**
Django [87] v.2.2	Salon Booking System Development
Django RESTframework [88]	Salon Booking REST API interfaces development
Python 3.5 [89]	Environment for running Django and Django RESTframework
Postgre SQL [90] v.13.0	Database for implementing functions of the Salon Booking system
Drawio [91]	Tool for drawing domain models
Eclipse Epsilon v.2.4 [92]	A set of tools for creating Ecore metamodels, models that conform to an Ecore metamodel and models to text transformation
JAVA JDK v.1.8.0_181 [93]	Environment for running the Epsilon tool
ADTool [94]	Attack Defense Tree tool for modelling and analysing attack defense scenarios represented by attack defense trees and attack defense terms

Table 8.1: Case Study Software Tools and Environment

library providing the necessary built in functionalities for developing APIs for any Django project. The Epsilon tool offers a wide range of tools for automating model management activities such as code generation and model management languages for; model to text transformation, model to model transformation and model validation all which extend the same core language. The Epsilon languages are abstracted from the

specifics of individual modelling technologies by a model connectivity layer which allows them to query and modify models that conform to different technologies in a uniform way. More details about the Epsilon tool can be accessed from here [95]. Drawio tool is adopted for visual construction of the domain models of the Salon Booking System. We use the ADTool for our threat modelling activities to model and analyze attack defense scenarios on our Salon Booking System APIs. These threat modelling activity had an end goal of generating some security requirements that were to be inter-weaved with their related functional requirements. More details about ADTool can be got from its project page [94].

8.2 The Case Study and Experiments of the Proposed Approach

To evaluate the effectiveness of our proposed approach, we applied our methodology to an empirical case study of a service based on an *Online Salon Booking System (OSBS)*. We first created RAML specifications that defined our OSBS service. The specifications formed a foundation from which we built a domain model representing our OSBS.

API EndPoint	**EndPoint Methods**
/salons	GET, POST
/salons/salon_id	GET, POST, DELETE
/salons/salon-services-categories	GET, POST, DELETE
/salons/salon-services-categories/category_id	GET, PATCH, DELETE
/salons/salon-services	GET, POST
/salons/salon-services/service_id	GET, PATCH, DELETE
/salons/customers	GET, POST
/salons/customers/salon_id/customer_id	GET, PATCH, DELETE
/salons/bookings	GET, POST
/salons/bookings/salon_id/booking_id	GET, POST, DELETE
/salons/stylists	GET, POST
/salons/stylists/salon_id/stylist_id	GET, POST, PATCH, DELETE

Table 8.2: Case Study API EndPoints

A total of 32 services covering the potential functions that may be used by the salon booking system were developed. Table 8.3 give the functional aspects covered by the services. The column *Salon API EndPoints* lists the prepared API endpoints that may be used by the Salon Booking System. The column *Numbers* summarizes the numbers of the different functional aspects provided by the case study API. We then modeled our domain entities which were salons, stylists, services and customers as *PrimaryResourceTypes*.

API Functions	**Salon API Endpoints**	**Numbers**
Top level API functions	ViewSalons, AddSalons	2
Single salon operations	ViewSalon, CreateSalon, DeleteSalon	3
Salon service categories	ListSalonCategories, CreateSalonCategory, DeleteCategories	3
Salon single category actions	ViewSalonCategory, UpdateSalonCategory, DeleteSalonCategory	3
Salon services actions	ListServices, AddServices	2
Salon single service actions	ViewSalonService, UpdateSalonService, DeleteSalonService	3
Salon customers operations	ViewCustomers, AddCustomers	2
Salon customer operations	ViewSalonCustomer, UpdateSalonCustomer, DeleteSalonCustomer	3
Salon bookings operations	ViewBookings, AddBookings	2
Individual salon bookings operations	ViewSingleBooking, AddSingleBooking, DeleteSingleBooking	3
Salon stylists operations	ViewStylists, AddStylists	2
Individual salon stylists operations	ViewSingleStylist, CreateStylist, UpdateStylist, DeleteStylist	3

Table 8.3: Case Study API Functional Aspects

Every salon, service, stylist, and customer has a name and relevant *Attributes*. We defined the attributes of each entity in our domain as value objects. For example, the salon entity had attributes such as *salon_id*, *name*, *phone*, *email*, and *address*. A customer entity had attributes such as *customer_id*, *name*, *phone* and *bio_description*. Next, we iteratively conducted threat modeling activities using ADTrees on our domain model entities. Our goal here was to identify potential vulnerabilities and attacks that could be leveraged on our API resources and their appropriate countermeasures. We defined the countermeasures as security requirements which we implicitly or explicitly interweave with their related functional requirement. For example, the *bio_description* attribute could have been assigned a string data type directly. However, upon using ADTree to model threats on an API function that would persist a customer entity data shape into our database, we identified that if the *bio_description* were left to accept any string data type, it could make its associated API endpoint vulnerable to stored XSS [96] attack. An attacker can include malicious script as part of the *bio_description* which would later be injected into a browser's DOM [96] and wreak havoc, whenever the affected customer entity data shape is retrieved by users with elevated privileges in our OSBS. By strictly defining a valid string representation of a *bio_description* with an invariant such as a custom string processing function that checks for any script tags or advanced XSS payloads not written in plain text such as base64 or binary, we can protect our API endpoint from stored XSS vulnerability.

We achieved this by moving *bio_description* attribute out of the customer entity and created a domain primitive value object *CustomerBio*. This implemented an invariant that protected our customer entity data shape from stored XSS vulnerability and its potential future mutations. Moreover, it enforced validation checks that could assert the validity of the *bio_description* value object for a certain operation while at the same time making it possible for our OSBS to perform any other specific action on it. This technique whenever applied in our threat modeling process achieved implicit interweaving of functional and security requirements. Some of the identified countermeasures constrained the behavior of our API end points without the need for defining them as domain primitives. These countermeasures whenever incorporated in our modelling process achieve the explicit interweaving of functional and security requirements. For example, to protect our OSBS endpoints that implement PUT, PATCH, POST and DELETE *MethodTypes*, from Cross Site Request Forgery attacks [97], we could provide a unique token among other required parameters for each API request. We enforced such type of constraints later when doing behavioral modeling using SOFL. We then created a metamodel representing our OSBS domain model by transforming our entities to *EClasses*, value objects and domain primitives to *EDataTypes*, and entity relations to *EReferences*. Figure 8.1 shows a sample generated Ecore metamodel of our OSBS even though we only showcase a few modeled entities for brevity and simplicity purposes. Finally, we generated formalized RESTful API specification via behavioral modeling of our designed metamodel. Here we transformed all the RESTful API methods for accessing

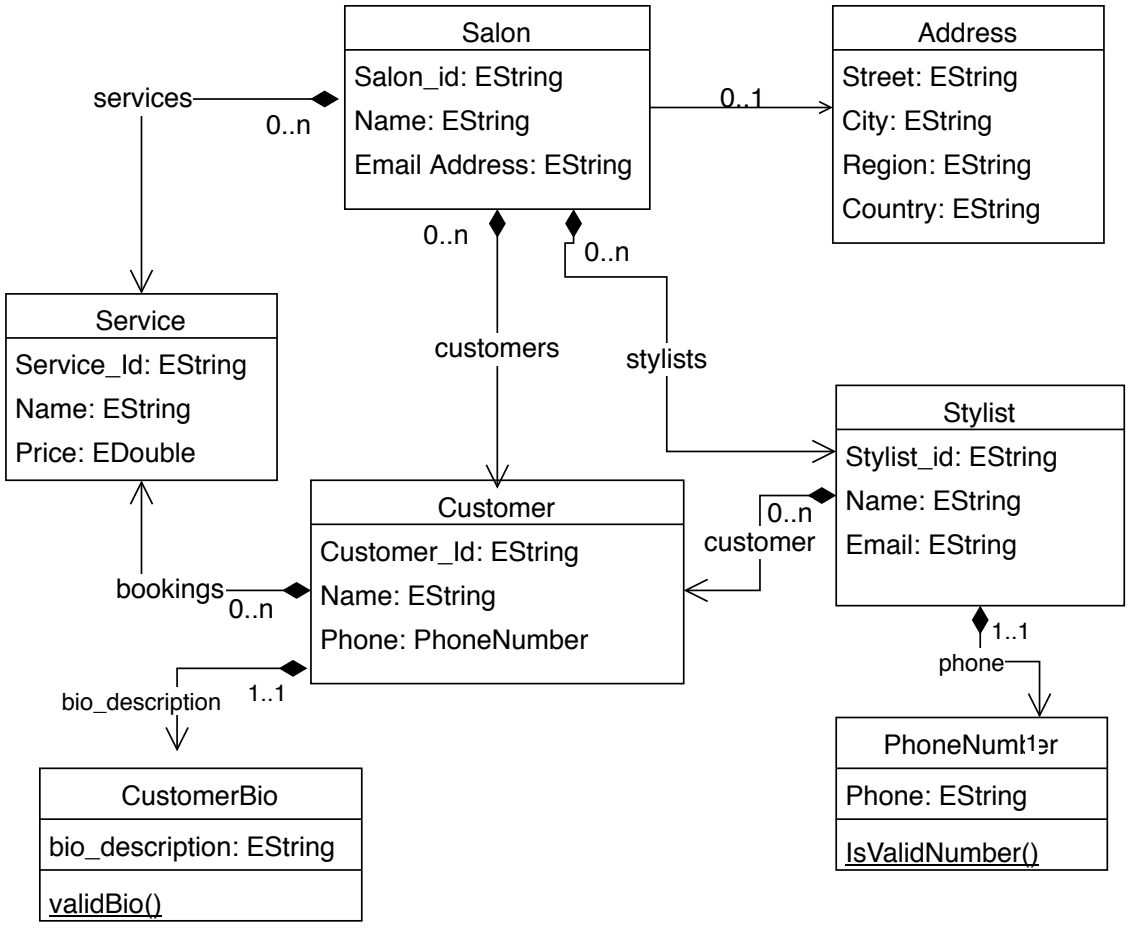

Figure 8.1: Sample Ecore metamodel for OSBS

the different *ResourceTypes* and *Collections* into SOFL processes with SOFL formal notation. A *Collection* represents a set of *ResourceTypes* exposed as XML or JSON. Listing 8.2 shows an excerpt of the end result of the API's SOFL formal specifications. For example, a RESTful API *UpdateAction* that updates the salon name of a single salon is transformed to a SOFL process *UpdateSalon* (lines 40-53), corresponding to PUT HTTP verb. The process relies on the output of a *Search* process that yields a *salon_object*. It then takes the *salon_object*, *new_salon_name* and a unique *token* as inputs, updates the salon's object attribute name and generates a status code that highlight success or failure of the process operation. The unique token represents a security requirement that constrains the behavior of the *UpdateSalon* process as a guard condition (line 52). SOFL offers its own type declarations therefore, the salon *ResourceType* (object) will be declared as of a *composite* type, *salon_id*, *new_salon_name* and unique *token* parameters as of a *string* type and the data store declared as *sequence* type to represent a *Collection* of salon *ResourceTypes*. In SOFL notation, a data store variable with a tilde sign e.g. ˜*salondatastore(i)* denotes the value of the data store before it is updated by a SOFL process. Table 8.4 gives a summary of a subset of all the process specifications of our case study, their number of functional scenarios and the test cases run for the functional scenarios. Rows 1 - 11 include process specifications which our proposed approach injected an access control security requirement to prevent Broken Object Level Authorization API vulnerability. A test relative to original RAML specification fails in the case where injected security measure (like requirement of an object level access control) is not respected, i.e., object level access control is not checked. Our generated SOFL specification correctly rejects such case (i.e., error message is returned), while the original RAML specification (incorrectly) dictates to accept such request, because it is not aware of such measure. A complete listing of all the process specifications with their functional scenarios can be accessed via this [1] full case study repository in the file *SOFL/salon_api_sofl_PFS.txt*.

[1] https://github.com/Egalaxykenya/IEICE-journal-paper-emeka

No.	Process Specification	Functional Scenarios	Test Cases	Passed Test Cases
1	*RetrieveSalon*	4	4	3
2	*AddSalon*	4	4	3
3	*DeleteSalon*	4	4	3
4	*GetSalonService*	4	4	3
5	*DeleteSalonService*	4	4	3
6	*GetSalonCustomer*	4	4	3
7	*AddSalonCustomer*	4	4	3
8	*DeleteSalonCustomer*	4	4	3
9	*GetSalonBookings*	4	4	3
10	*GetSalonBooking*	4	4	3
11	*DeleteSalonBooking*	4	4	3
12	*AddSalonServiceCategory*	3	3	3
13	*GetSalonServiceCategories*	3	3	3
14	*GetSalonServiceCategory*	3	3	3
15	*DeleteSalonServiceCategory*	3	3	3
16	*GetSaloncategoryServices*	3	3	3
17	*CreateSalonService*	3	3	3
18	*RetrieveSalons*	3	3	3
19	*GetSalonCustomers*	3	3	3

Table 8.4: Summary of case study process specifications

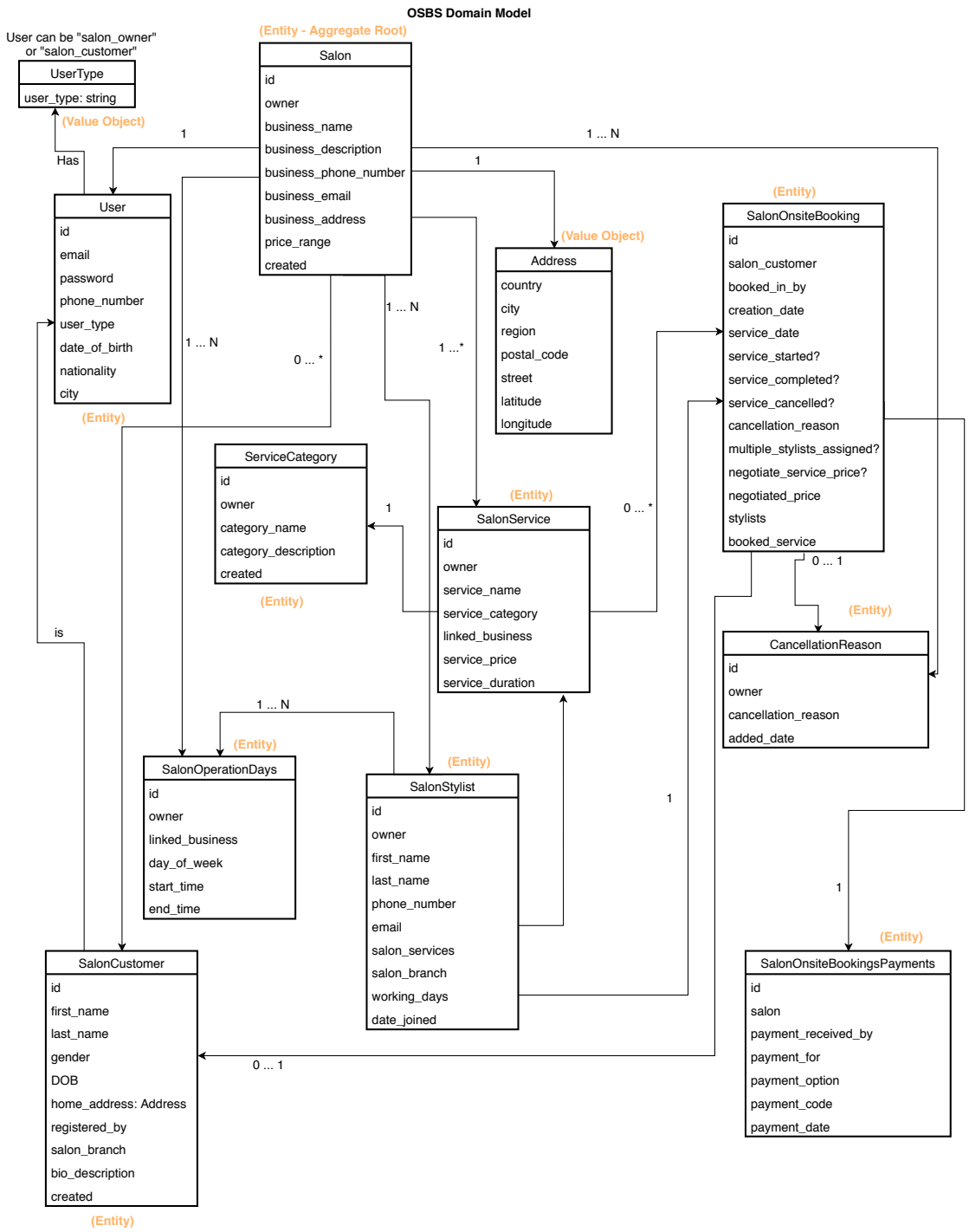

Figure 8.2: Salon Booking System Domain Model

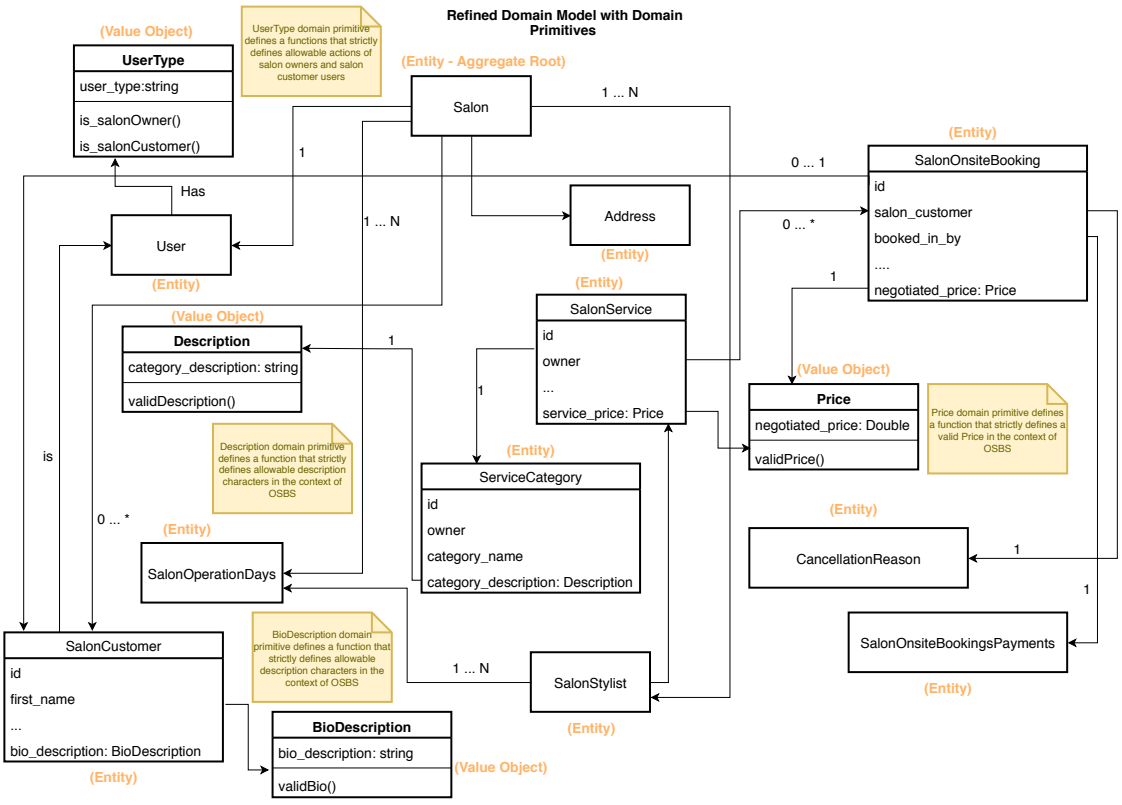

Figure 8.3: Salon Booking System Refined Domain Model

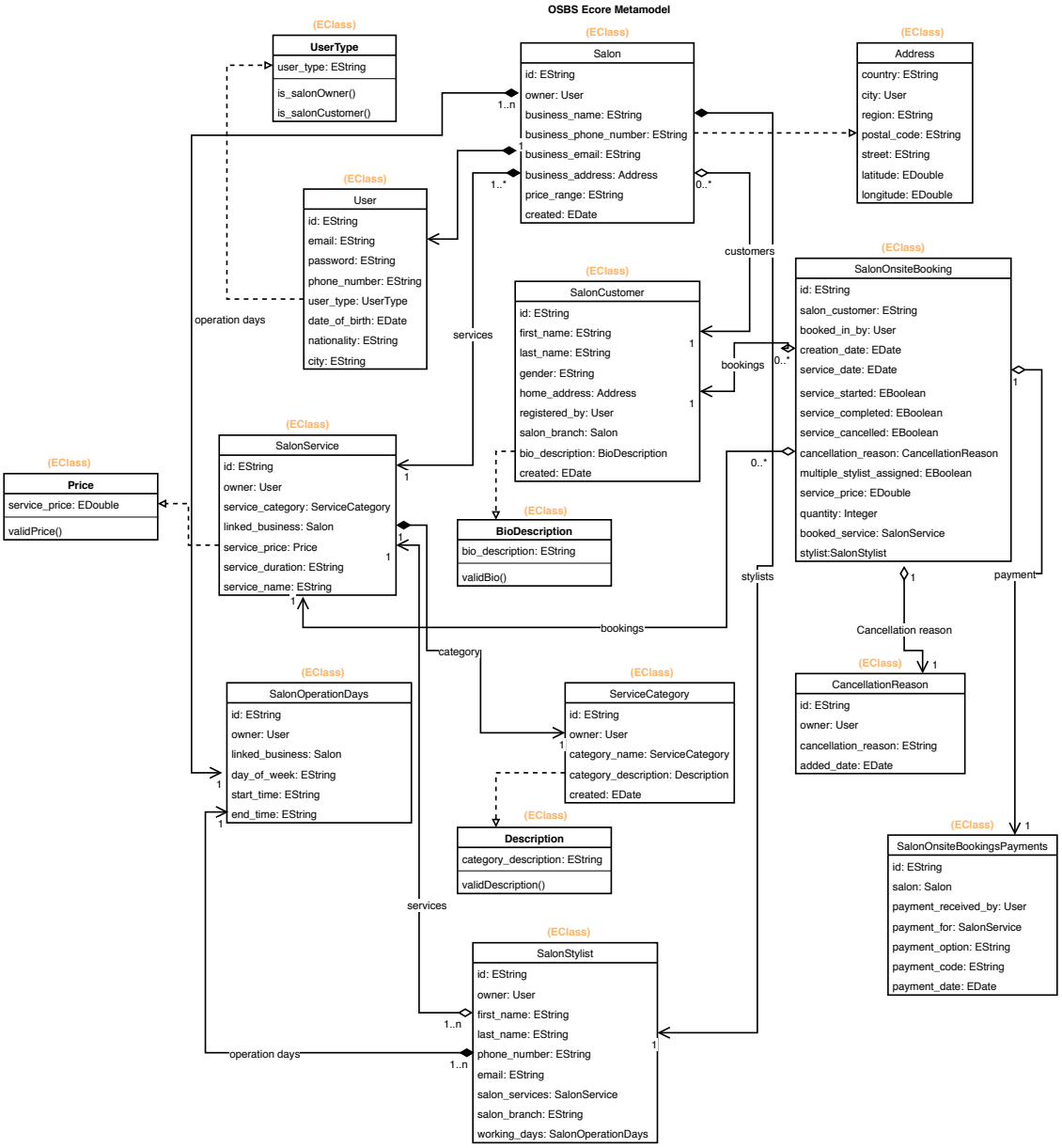

Figure 8.4: Salon Booking System Meta Model

```
#% RAML 1.0
title: SalonService API
baseUri: http://localhost:8000/api/{version}
version: v1
mediaType: application/json

uses:
shapes: ./dataTypes/shapes.raml

resourceTypes:
collection: !include resourceTypes/collection.raml

securitySchemes:
oauth_2_0: !include securitySchemes/oauth2_0.raml

/salons:
   type:
    collection:
       response-type: shapes.SalonData[]
       request-type: shapes.NewSalonRequestData
   get:
      description: Get a list of Salons based on the salon name
      queryParameters:
        salon_name:
          displayName: Salon Name
          type: string
          description: Salon's name
          example: "Salon Paradise"
          required: true
         ...

    post:
       description: Salon data created correctly for salon business
       body: shapes.NewSalonRequestData
    delete:
     ...
    /{salon_id}:
       type: ...
       get:
        description: Get the salon with `salon_id = {salon_id}`
         responses:
          200:
           body:
            application/json:
             example: |
              {
               "data": {
               "id": "lsVx",
               "name": "Salon Paradise",
               "location": "Chuo Ku,Tokyo",
               "link":"http://localhost:8000/api/v1/salons/SalonParadise"
             },
             "success": true,
             "status": 200
            }
               ...
```

Listing 8.1: Sample RAML description file

```
module Salon_API;

    class Address;

        var
        street, city, region, country: string

        method Init()

            post street = "" and city = "" and country = ""
                        and region = ""

        end_method;

    end_class;

    type

    SalonData = composed of
                          salon_id = string
                          name: string
                          email: string
                          address: Address
                end;
    SalonCollection = seq of SalonData;
    var salon_datastore: SalonCollection

    inv /* salon_id uniquely identifies SalonData in salon_datastore */
    forall[i,j: inds(salon_datastore)] | i <> j
        => salon_datastore(i).salon_id
          <> salon_datastore(j).salon_id;

    process Search(search_id: string) salon_object: SalonData
        ext rd salon_datastore
        pre exists([i: inds(salon_datastore)] |
                 salon_datastore(i).salon_id = search_id)
        post salon_object inset elems(salon_datastore) and salon_object = search_id
    end_process;

    process UpdateSalon(salon_object: SalonData, validtoken: string,
                        salon_name: string) status_message: string
        ext wr salon_datastore
        pre exists([i: inds(salon_datastore])] |
                 salon_datastore(i) = salon_object)
        post len(salon_datastore) = len(~salon_datastore)
           and (forall[i:inds(~salon_datastore)] |
                (~salon_datastore(i) = salon_object
           => salon_datastore(i)
               = modify(~salon_datastore(i), name-->salon_name))
            and (~salon_datastore(i) <> salon_object
          => salon_datastore(i) = ~salon_datastore(i)))
        and validtoken <> "" and status_message = "HTTP 204"
    end_process;
end_module;
```

Listing 8.2: SOFL formal specification for RESTful API UpdateAction

Chapter9

Conclusion

The increasing demand for efficient, secure and interconnected systems powering complex business requirements have elicited great interests in both the industry and research communities. Development methods and frameworks for designing, developing and deploying reliable and secure APIs are one of the most crucial topics in software engineering. While the spread of APIs is driving more sophisticated applications that enhances and amplify our own abilities, their is a flip side of increased risks as we become more dependent on APIs for our critical system tasks. The more we use APIs, the greater their potential to be attacked.

With a focus on designing, modelling and developing APIs exposed over HTTP using RESTful approach, we propose a model driven formal engineering approach to the design and development of security aware RESTful web APIs in this dissertation. Specifically, the model driven formal engineering approach supports the formal specification construction using the SOFL specifications language and the formal specification based testing approach via a rigorous testing procedure.

We conducted an empirical case study by developing a Salon booking system to validate the feasibility of our proposed model driven approach to the design of security aware RESTful web APIs as well as offered a mechanism for generating SOFL specifications from a domain model via model to text transformation. Specifically, our research and its novelty are summarized as follows:

9.1 Model Driven Formal Engineering Approach for Security Aware RESTful Web APIs

We have presented a novel model driven approach for design and development of security aware RESTful web APIs by providing mechanisms for interweaving both functional and security requirements. It tackles the most fundamental challenge in the area of RESTful API design that is how to interweave security requirements and functional requirements in the modelling process of building secure, robust and reliable RESTful APIs. It also saves cost and efforts by leveraging on existing methodologies, techniques and support tools in API design and development. The main contribution of our proposed model driven formal engineering approach for design and development of secure RESTful web APIs is a systematic framework that offers modelling through a four step approach covering both structural and behavioural modelling of APIs with a focus on security. Our proposed approach suggests that security requirements can be incorporated early into

the API design process during the domain modelling stage. That is, security requirements are constructed as domain primitives in our domain model during the threat modelling process using attack defense trees threat modelling technique.

The satisfiability of these API security requirements with respect to their related functional requirements are checked via the specification testing. Compared with the related approaches (see Chapter 4) to security aware requirements specification techniques, our proposed model driven formal engineering approach offers a four step technique for constructing security aware formal RESTful API specification from scratch. Specifically, our techniques combines the principles of domain primitives in Domain Driven Design, attack defense trees in threat modelling, SOFL formal engineering techniques and model to text transformations. The proposed framework can be applied for interweaving security and functional requirements in the design of RESTful web APIs allowing for security requirements to be given an equal focus as functional requirements during the API design and development process thus encouraging the production of secure APIs.

9.2 API formal Specification generation via Model to Text Transformation

To facilitate practical generation of SOFL formal specifications of RESTful APIs, we have proposed a technique for generating SOFL formal specifications from domain models using Model to Text transformation and Epsilon generation language. Specifically, we create a meta model that our source domain model corresponds to then using the Epsilon Generation Language write rules that generate SOFL module specification minus the pre- and post conditions declarations in the module processes. The pre- and post condition processes are manually filled to generate the final formal specifications.

The most distinguished merit of the model to text transformation is the utilization of the API's domain model as well as a metamodel that the domain model corresponds to as the foundation for generation of formal SOFL specifications that is a representation of API's functional and security requirements, and the preservation of statically generated text which encourages synchronization on changes on the source model and the generated SOFL specifications.

Chapter10

Future Work

In this chapter, we focus on the discussion of our future research plans.

10.1 Enhancement of the Proposed Approach

Our proposed model driven formal engineering technique offers a 6 step approach, that aims at providing a secure by design approach to API design by encouraging the interweaving of both functional and security requirements. The enhancement of the techniques especially on the automated generation of domain models form the input source RAML document as well as automated generation of pre-/post conditions in the generated SOFL formal specifications as a result of the model to text transformations are considered areas of future research.

10.2 Enhancement of the Specification Testing

One major concern of specification testing is the efficiency of generating functional scenario sequences. Based on this approach, all possible functional scenarios need to be generated. As a result, you may have instances where the number of functional scenarios is very large is the involved API process consists of many functional scenarios. The large number of functional scenario sequences may pose a difficulty in checking the infeasible functional scenario sequences. A formal technique on how to appropriately construct the functional scenarios sequences is an interesting area in our future research work.

10.3 Development of a Supporting Tool for Domain Models Construction and Importation of security constructs from ADTrees

A supporting tool to supplement our proposed approach is critical as some activities such as creation of domain models from API graphs and injection of security attributes from ADtree analysis may currently require a considerable amount of human effort and time. Most importantly, automating steps that involves manual user interaction such as the construction of domain models from the generated API resource graph in

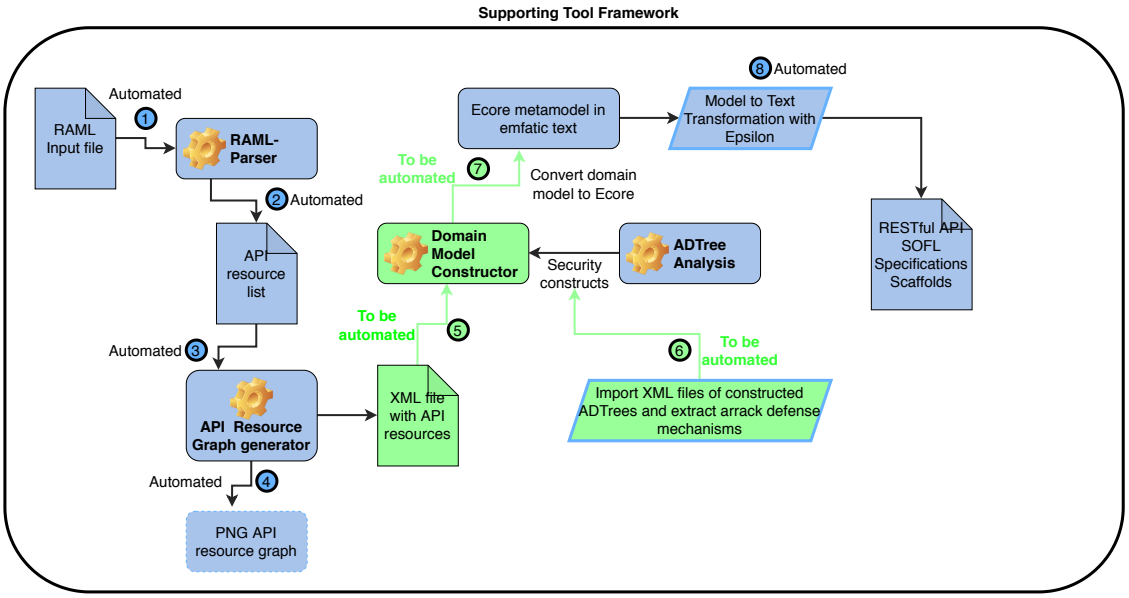

Figure 10.1: Overall Framework with Sections for future work

step 2, which is currently semi-automatic will considerably reduce the human effort required in these steps. Our future work seeks to automate this by building an integrated tool that could leverage on consuming inputs from existing tools such as ADTree tool for construction of ADtrees, Epsilon tool for Model to Text transformation and RAML parser. Figure 10.1 showcases the integrated framework that seek to achieve automation of steps currently done manually using our proposed approach. Solid green arrows indicate processes which are currently achieved manually. We plan to extend our currently developed API resource graph generator script to provide an XML version of resource graph generator which can be consumed by a domain model constructor. Similary, we plan to make the domain model constructor consume an XML file generated by the current ADTree tool and filter out defense mechanisms which are injected into the target domain model as security constructs. Furthermore, we plan to automate the construction of an Ecore metamodel by making our domain model constructor yield an *emfatic* textual output of a metamodel that a RESTful API domain model corresponds to.

The development of this supporting tool will go a long way in providing a powerful integrated environment to support more intelligent, effective and efficient development of secure RESTful web APIs. We hope our research work of a model driven formal engineering approach to secure design of RESTful web APIs will be more mature for large-scale and complex API design, development and deployment through our continuous research efforts in the near future.

AppendixA

Appendix

A.1 List of Published Papers

Refereed Journal Papers (First Author)

- Busalire Onesmus Emeka, Soichiro. Hidaka, Shaoying. Liu, "A Practical Model Driven Approach for Designing Security Aware RESTful Web APIs using SOFL", IEICE TRANSACTIONS on Information and Systems, Vol. 2, E106-D,No.5,pp.-,May. 2023

Refereed International Conference Papers (First Author)

- Busalire Onesmus Emeka, Soichiro. Hidaka, Shaoying. Liu, "A Formal Approach to Secure Design of RESTful Web APIs using SOFL", in The 10th International Workshop on SOFL + MSVL for Reliability and Security, 2020, Singapore, March 1, 2021, doi: 10.1007/978-3-030-77474-5_8 Publisher: Springer International Publishing. Part of the Lecture Notes in Computer Science book series (LNCS, volume 12723, pages 105 - 125)

- Busalire Onesmus Emeka and Shaoying. Liu, "A Formal Technique for Concurrent Generation of Software's Functional and Security Requirements in SOFL Specification", in The 9th International Workshop on SOFL + MSVL for Reliability and Security (SOFL+MSVL 2019) in Shenzhen, China, November 5, 2019, doi: 10.1007/978-3-030-41418-4_2. Co-located with The 21st International Conference on Formal Engineering Methods ICFEM 2019 Publisher: Springer International Publishing Part of the Lecture Notes in Computer Science book series (LNCS, volume 12028, pages 13-28)

- Busalire Onesmus Emeka and Shaoying. Liu, "Assessing and extracting software security vulnerabilities in SOFL formal specifications", in 2018 International Conference on Electronics, Information, and Communication (ICEIC), Honolulu, Hawaii, 24-24 Jan 2018, pp.374-377, doi: 10.23919/ELINFOCOM.2018.8330613. Conference Paper. Publisher: IEEE

- Busalire Onesmus Emeka and Shaoying. Liu, "Security Requirement Engineering Using Structured Object-Oriented Formal Language for M-Banking Applications," 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), Prague, Czech Republic, 25-29 July 2017, pp. 176-183, doi: 10.1109/QRS.2017.28. Short Paper. Publisher: IEEE

Technical Workshops (First Author)

- Busalire Onesmus Emeka, Soichiro. Hidaka, Shaoying. Liu, "Semi-automatic trans-formation of ecore metamodels to sofl based specifications," PSJ 211th Meeting of Special Interest Group on Software Engineering, IPSJ Technical Reports, Vol. 022-SE-211, No.9, Jul. 2022

A.2 Case Study SOFL Specifications

```
1
2 /* SOFL Implicit Formal Specifications for Salon API project */
3 /* Begin class declarations */
4
5 class SalonUser;
6 type
7 Year = string;
8 Month = string;
9 Day = string;
10 DOB = Year * Month * Day;
11 UserRoles = {<SALONOWNER>, <SALONCUSTOMER>} /* Set enumerating user types in the context of the salon booking system */
12 User_Selection = {<owner>, <customer>}
13
14
15 var
16 id: string
17 email: string
18 password: string
19 date_of_birth: DOB
20 nationality: string
21 city: string
22 role: UserRoles
23
24
25 inv
26
27 len(id) == 36; /* define ids as of standard UUID string type with length of 36 characters */
28
29 method Init()
30   post id = " " and email = " " and password = " " and date_of_birth = nil and nationality = " " and city = " " and role = {}
31
32 end_method;
33
34 method Set_SalonUserAttributes(id: string, email: string, password: string date_of_birth: DOB, nationality: string, city: string)
35     post id=id and email=email and password=password and date_of_birth=date_of_birth and nationality=nationality and city=city
36
37 end_method;
38
```

```
method Set_SalonUserRole(role_selection: User_Selection)
  explicit
    begin
      post case role_selection of
            <owner>-->this.role := <SALONOWNER>
            <customer>-->this.role := <SALONCUSTOMER>
          end_case
    end
end_method

end_class;

class BioDescription;

var
bio_description: string
invalid_chars: string
inv
0 < len(bio_description) <= 100

method Init()
  post bio_description = Nil
  invalid_chars = ['<','>', ';', '&', '#', '=','/', '\', '%']
end_method;

method Set_Bio_Description(bio_description: string)
  post bio_description = bio_description
end_method;

method validBio(invalid_chars: string, bio_description: string) is_valid: boolean
/* This method tests for existence of invalid characters in the provided bio_description string */
  pre elems(invalid_chars) <> {}
  post is_valid = true and inter(elems(invalid_chars), elems(bio_description)) = {} or
       is_valid = false and inter(elems(invalid_chars), elems(bio_descriotion)) <> {}
end_method;

end_class;

class Price;

var
service_price: nat0

inv
0 < service_price <= 1000000
method Init()
  post service_price = 0
end_method;

```

```
method validPrice(price: nat0) service_price: nat0
  post service_price = price
              and service_price > 0 and service_price <= 100000
end_method;

end_class;

/* End class declarations */


module Salon_API;

type
Year = string;
Month = string;
Day = string;
DOB = Year * Month * Day;

hour = string;
minutes = string;
token = string;
User_Selection = {<owner>, <customer>}

Timestamp = Year * Month * Day * hour * minutes;

AddressData = composed of
                address_country = string
                address_city = string
                address_region = string
                postal_code = string
                address_street = string
                latitude = nat0
                longitude = nat0
              end;


SalonData = composed of
              id = string
              owner = SalonUser
              business_name = string
              business_type = string
              business_description = string
              business_phone_number = string
              business_email = string
              business_address = AddressData
              price_range = string
              created = Timestamp
            end;

```

```
SalonCustomerProfile = composed of
                id = string
                user = SalonUser
                name = string
                phone_number = string
                bio_description = BioDescription
              end;


SalonServiceCategoryData = composed of
                            id = nat0
                            owner = SalonUser
                            category_name = string
                            created = Timestamp
                          end;

SalonServiceData = composed of
                    id = string
                    owner = SalonUser
                    service_category = SalonServiceCategoryData
                    linked_business = SalonData
                    service_price = Price
                    service_duration = string
                    service_name = string
                  end;

SalonServiceCancellationReasonData = composed of
                                      id = string
                                      owner = SalonUser
                                      cancellation_reason = string
                                      added_date = Timestamp
                                    end;

SalonBookingData = composed of
                    id = string
                    salon_customer = SalonUser
                    booked_in_by = SalonUser
                    stylist = SalonUser
                    booked_service = SalonServiceData
                    booking_creation_date: Timestamp
                    service_date: Timestamp
                    service_started = bool
                    service_completed = bool
                    service_cancelled = bool
                    cancellation_reason = SalonServiceCancellationReasonData
                    negotiate_service_price = bool
                    negotiated_price = Price
                  end;

```

```
SalonOperationDaysData = composed of
                          id = string
                          owner = SalonUser
                          linked_business = SalonData
                          day_of_week = string
                          start_time = Timestamp
                          end_time = Timestamp
                        end;

SalonStylistData = composed of
                     id = string
                     owner = SalonUser
                     first_name = string
                     last_name = string
                     phone_number = string
                     email = string
                     salon_service = SalonServiceData
                     salon_branch = SalonData
                     working_days = SalonOperationDaysData
                     date_joined = Timestamp
                   end;

SalonTable = map SalonData to SalonUser;
SalonServicesCategoriesTable = seq of SalonServiceCategoryData;
SalonServiceTable = map SalonServiceData to SalonServiceCategoryData;
MapSalonServiceToSalon = map SalonServiceData to SalonData;
MapSalonServiceCategoryToSalon = map SalonServiceCategoryData to SalonData
MapSalonCustomerToSalon =  map SalonUser to SalonData;
MapSalonBookingsToCustomer = map SalonBookingData to SalonUser;
MapSalonCustomerProfileToSalonUser = map SalonCustomerProfile to SalonUser;

var

/* External data stores for salon records */
#salons_table: SalonTable
#salon_services: SalonServiceTable
#single_salon_services: MapSalonServiceToSalon
#salon_services_categories: MapSalonServiceCategoryToSalon
#service_categories: SalonServiceCategoriesTable
#salon_customers_table: MapSalonCustomerToSalon
#salon_bookings_table: MapSalonBookingsToCustomer
#salon_stylist_table: map SalonStylistData to SalonData;
#salon_users_table: seq of SalonUsers;
#salon_customer_profile: MapSalonCustomerProfileToSalonUser


inv
/* Each salon category is unique
  Each user is unique and has a unique id
```

```
235 */
236 forall[i,j: inds(salon_services_categories)] | i <> j => salon_services_categories(i).id <> salon_services_categories(j).id;
237 forall [i,j: inds(salon_users_table)] | i <> j => salon_users_table(i).id <> salon_users_table(j).id
238
239
240
241 process CreateSalonUser(validtoken: token, access_token: token, id: string, email: string, date_of_birth: DOB, password: string,
242     nationality: string, city: string, role_selection: User_Selection ) salonuser: SalonUser, response_status: string | error_message: string
243 ext wr salon_users_table
244 pre not exists[i: inset salon_users_table] | i.id = id
245 explicit
246   begin
247     if access_token = validtoken and access_token <> nil and elems(access_token) <> {}
248     then
249       salonuser := new SalonUser
250       salonuser.SetSalonUserAttributes(id, email, password, date_of_birth, nationality, city)
251       salonuser.Set_SalonUserRole(role_selection)
252       salon_users_table := conc(~salon_users_table, [salonuser])
253       response_status := "Http 200 Ok"
254     else
255       error_message := "Http 401, Permission Denied"
256   end
257 end_process;
258
259 process CreateSalonCustomerProfile(validtoken: token, access_token: token, id: string, user: SalonUser, name: string, phone_number: string,
260     bio_description: BioDescription, bio: string) customer_profile: SalonCustomerProfile, error_message: string
261 ext rd salon_users_table
262 ext wr salon_customer_profile
263 pre not exists[i: inset dom(salon_customer_profile)] | i.id = id
264
265 explicit
266   begin
267     if access_token = validtoken and access_token <> nil and elems(access_token) <> {}
268     then
269       bio_description := new BioDescription
270       validbio = bio_description.validBio(bio)
271       if validbio = true and user inset(elems(salon_users_table))
272         then bio_description.Set_Bio_Description(bio)
273           customer_profile = mk_SalonCustomerProfile(id, user, name, phone_number, bio_description)
274           salon_customer_profile = override(~salon_customer_profile, {customer_profile-->user})
275       else error_message := "Bio description contains invalid characters";
276     else error_message := "Http 402, Permission Denied".
277
278   end
279
280 end_process
281
282 process RetrieveSalon(validtoken: token, access_token: token, salon_id: string, salonuser: SalonUser)
283     salon: SalonData, response_status: string | error_message: string
```

```
/* API resource path: /salons/{salon_id} */

ext rd salons_table
ext rd salon_users_table
pre exists[i: inset dom(salons_table)] | i.id = salon_id
                and salons_table(i) = salonuser
                  and salonuser inset(elems(salon_users_table))

post exists![i: inset dom(salons_table)] | i.id = salon_id and i = salon and salons_table(i) = salonuser  and salonuser.role = <SALONOWNER>
                and access_token = validtoken
                  and len(access_token) <> 0
                      and access_token <> nil
                        and salonuser inset(elems(salon_users_table))
                          and salons_table(salon) = salonuser
                            and salon.id = salon_id
                              and response_status = "Http 200 Ok"
                or
                error_message = "Http 401, Permission Denied"
                  and access_token <> validtoken
                    and len(access_token) = 0
                        and access_token = nil
                          and exists![i: inset dom(salons_table)] | i.id = salon_id | salons_table(i) = salonuser and salonuser.role = <SALONOWNER>
                            and salonuser inset(elems(salon_users_table))

end_process;

process RetrieveSalons(validtoken: token, access_token: token, resource_path: string) salons: SalonTable, response_status: string | error_message: string
/* API resource path: /salons */
ext rd salons_table
pre elem(resource_path) <> {}
post salons = salons_table
                and access_token = validtoken
                  and elems(access_token) <> {}
                      and access_token <> nil
                        and elems(resource_path) <> {}
                          and response_status ="Http 200 Ok"
                 or
                  error_message = "Http 401, Permission Denied"
                    and access_token <> validtoken
                      and elems(access_token) = {}
                        and access_token = nil
                          and elems(resource_path) <> {}

end_process;

process AddSalon(validtoken: token, access_token:token, salon: SalonData, owner: SalonUser) added_salon: SalonData,
    response_status: string | error_message: string
/* API resource path: /salons */
ext wr salons_table
```

```
ext rd salon_users_table

pre not exists[i: inset dom(salons_table) ] | i = salon  | salons_table(i) = owner
                and owner inset elems(salon_users_table)

post  exists[i: inset dom(salons_table)] | i = added_salon and salons_table(i) = owner and owner.role = <SALONOWNER>
                => access_token = validtoken
                  and len(access_token) <> 0
                      and access_token <> nil
                        and salons_table = override(~salons_table, {salon-->owner})
                          and added_salon inset dom(salons_table)
                            and salons_table(added_salon) = owner
                              and salons_table(added_salon).role = <SALONOWNER>
                                and salons_table <> ~salons_table
                                  and response_status = "Http 200 Ok"
                or
                error_message = "Http 401, Permission Denied"
                  and access_token <> validtoken
                    and len(access_token) = 0
                        and access_token = nil
                          and not exists[i: inset dom(salons_table) ] | i = salon  | salons_table(i) = owner and owner.role = <SALONOWNER>
                              and owner inset elems(salon_users_table)

end_process;

process DeleteSalon(validtoken: token, access_token:token, salon_id=string, owner: SalonUser) deleted_salon: SalonData,
    response_status: string | error_message: string
/* API resource path: /salons/{salon_id} */

ext wr salons_table
ext rd salon_users_table

pre exists![i: inset dom(salons_table)] | i.id = salon_id and salons_table(i) = owner and salons_table(i).role = <SALONOWNER>
post ~salons_table = override(salons_table, {deleted_salon-->owner})
                  => access_token = validtoken,
                    and len(access_token) <> 0
                      and access_token <> nil
                        and exists![i: inset dom(~salons_table)] | i.id = salon_id and ~salons_table(i) = owner and owner.role = <SALONOWNER>
                          and deleted_salon.owner = owner
                            and owner inset elems(salon_users_table)
                              and owner.role = <SALONOWNER>
                                and deleted_salon.id = salon_id
                                  and salons_table <> ~salon_table
                                    and response_status = "Http 204"
                or
                 error_message = "Http 401, Permission Denied"
                  and access_token <> validtoken
                    and len(access_token) = 0
                        and access_token = nil
```

```
382                         and exists![i: inset dom(salons_table)] | i.id = salon_id | salons_table(i) = owner and owner.role = <SALONOWNER>
383                           and owner inset elems(salon_users_table)
384                             and owner.role = <SALONOWNER>
385
386 end_process;
387
388 process AddSalonServiceCategory(validtoken: token, access_token:token, category: SalonServiceCategoryData) added_category: SalonServiceCategoryData,
389     response_status: string | error_message: string
390
391 /* API resource path: /salons/salon-services-categories */
392 ext wr salon_services_categories
393 pre not exists[i: inds(salon_services_categories)] | salon_services_categories(i).id = category.id
394 post salon_services_categories = conc(~salon_services_categories, [category])
395                 => access_token = validtoken
396                   and len(access_token) <> 0
397                       and access_token <> nil
398                         and added_category.id = category.id
399                           and added_category inset elems(salon_services_categories)
400                             and len(~salon_services_categories) <> len(salon_services_categories)
401                               and response_status = "Http 200 Ok"
402                 or
403                 error_message = "Http 401, Permission Denied"
404                   and access_token <> validtoken
405                     and len(access_token) = 0
406                         and access_token = nil
407
408
409 end_process;
410
411 process GetSalonServiceCategories(resource_path: string, salon_id: string, validtoken: token, access_token: token) categories: MapSalonServiceCategoryToSalon,
412     response_status: string | error_message: string
413 /* API resource path: /salons/{salon_id}/salon-services-categories */
414 ext rd salon_services_categories
415 pre elems(request_path) <> {}
416 post forall([x: inset dom(categories)] | subset({x}, dom(salon_services_categories)) <=> true and categories(x).id = salon_id and subset(dom(categories),
417     dom(salon_services_categories)) <=> true)
418                 => access_token = validtoken
419                   and len(access_token) <> 0
420                       and access_token <> nil
421                         and elems(resource_path) <> {}
422                           and response_status = "Http 200 Ok"
423                 or
424                   error_message = "Http 401, Permission Denied"
425                     and access_token <> validtoken
426                       and len(access_token) = 0
427                           and access_token = nil
428
429 end_process;
430
```

```
process GetSalonServiceCategory(validtoken: token, access_token: token, salon_id: string, category_id: string)
    service_category: SalonServiceCategoryData, response_message: string | error_message: string
/* API resource path: /salons/{salon_id}/salon-services-categories/{category_id} */

ext rd salon_services_categories
pre exists[i: inset dom(salon_services_categories)] | i.id = category_id and salon_services_categores(i).id = category_id
post exists[i: inset dom(salon_services_categories)] | i = service_category and i.id = category_id
    and salon_services_categories(service_category).id = salon_id
               => access_token = validtoken
                 and len(access_token) <> 0
                     and access_token <> nil
                       and exists![i: inset dom(salon_services_categories)] | i = service_category and i.id = category_id
                           and salon_services_categories(i).id = salon_id
                           and response_message = "Http 200 Ok"
               or
                 error_message="Http 401, Permission Denied"
                   and access_token <> validtoken
                     and len(access_token) = 0
                         and access_token = nil
                           and exists![i: inset dom(salon_services_categories)] | i = service_category and i.id = category_id
                               and salon_services_categores(i).id = salon_id

end_process;


process DeleteSalonServiceCategory(validtoken: token, access_token: token, salon_id:string category_id: string) deleted_category: SalonServiceCategoryData,
    response_status: string | error_message: string
/* API resource path: /salons/{salon_id}/salon-services-categories/{category_id} */
ext wr service_categories
pre exists[i: inds(service_categories)] | service_categories(i).id = category_id and service_categories(i).linked_business.id = salon_id
post not exists[i: inds(services_categories)] | service_categories(i).id = category_id | service_categories(i) = deleted_category
            => access_token = validtoken
              and len(access_token) <> 0
                  and access_token <> nil
                    and service_categories = conc(~service_categories, [deleted_category])
                      and len(service_categories) <> len(~service_categories)
                         and deleted_category.linked_business.id = salon_id
                            and response_status = "Http 204")
            or
              error_message = "Http 401, Permission Denied"
                and access_token <> validtoken
                  and len(access_token) = 0
                      and access_token = nil
                        and exists![i: inds(service_categories)] | service_categories(i).id = category_id
                            and  service_categories(i).linked_business.id = salon_id

end_process;

process GetSalonCategoryServices(validtoken: token, access_token: token, salon_id, category_id: string) service_list: SalonServiceTable,
```

```
    response_message: string | error_message: string
/* API resource path: /salons/{salon_id}/salon-services-categories/{category_id}/salon-services
Get specific salon services for a specific service category
 */
ext rd salon_services
ext rd single_salon_services
pre exists![i: inset dom(salon_services)] salon_services(i).id = category_id and salon_services(i).linked_business.id = salon_id
post subset(dom(service_list), dom(salon_services) <=> true) and forall[x: inset dom(salon_services)] | salon_services(x).id = category_id
    and salon_services(x).linked_business.id = salon_id
                => access_token = validtoken
                and len(access_token) <> 0 and access_token <> nil
                  and exists![i: inset dom(salon_services)] | salon_services(i).id = category_id
                    and exists![j: inset dom(single_salon_services)] | single_salon_services(j).id = salon_id
                      and response_message = "Http 200 Ok"
                or
                error_message = "Http 401, Permission Denied"
                  => access_token <> validtoken
                  and len(access_token) = 0
                    and access_token = nil
                      and exists![i: inset dom(salon_services)] | salon_services(i).id = category_id and salon_services(i).linked_business.id = salon_id

end_process;

process CreateSalonService(valdtoken: token, access_token: token, user: SalonUser, category: SalonServiceCategoryData,
    salon: SalonData, salon_service: SalonServiceData) created_service: SalonServiceData, response_status: string | error_message
/*
Create a salon service for a specific salon
 */
ext rd salons_table
ext rd salon_services_categories
ext wr salon_services
ext wr single_salon_services

pre not exists[i: inset dom(single_salon_services)] | i = salon_service and i.id = salon_service.id
post single_salon_services = override(~single_salon_services, {salon_service-->salon})
            and single_salon_services(created_service) = salon
                and single_salon_services <> ~single_salon_services
                  and category inset(elems(salon_services_categories))
                    and salon_services = override(salon_services, {salon_service-->category}
                      and salon_services <> ~salon_services
                        and exists[i: inset dom(single_salon_service)] | i = created_service and single_salon_service(i) = salon
                          and exists[j: inset dom(salon_service)] | salon_services(j) = category
                            and response_status = "Http 200 Ok"
          => access_token = validtoken
            and len(access_token) <> 0
                and access_token <> nil
                  and salons_table(salon) = user
                    and user.role = <SALONOWNER>
      or
```

```
        error_message = "Http 401, Permission Denied"
          and access_token <> validtoken
            and len(access_token) = 0
              and access_token = nil
                and salons_table(salon) = user
                 and user.role = <SALONOWNER>

end_process;

process GetSalonService(validtoken: token, access_token: token, salon_id: string, service_id: string, salonuser: SalonUser)
    salon_service: SalonServiceData, response_status: string | error_message: string
/* API resource path: /salons/{salon_id}/salon-services/{service_id}
Get salon services for a specific salon owned by a given user
 */

ext rd single_salon_services
ext rd salons_table

pre exists[i: inset dom(single_salon_services)] | i.id = service_id

post exists![i: inset dom(single_salon_services)] | single_salon_services(i).id = service_id and i.id = service_id
                and salon_service.id = service_id
                    and response_status = "Http 200"
                => access_token = validtoken
                  and len(access_token) <> 0
                    and access_token <> nil
                      and exists[j: inset dom(salons_table)] | j.id = salon_id and salons_table(j) = salonuser
                        and salons_table(salon) = salonuser
                          and salonuser.role = <SALONOWNER>
            or
              error_message = "Http 401, Permission Denied"
              and access_token <> validtoken
                and len(access_token) = 0
                  and access_token = nil
                    and salons_table(salon) = salonuser
                      and exists[j: inset dom(salons_table)] | j.id = salon_id and salons_table(j) = salonuser
                        and salonuser.role = <SALONOWNER>

end_process;

process DeleteSalonService(validtoken: token, access_token: token, salon_id: string, service_id: string, salonuser: SalonUser)
    deleted_service: SalonServiceData, response_status: string | error_message: string

/* API resource path: /salons/{salon_id}/salon-services/{service_id}
ext rd single_salon_services
ext rd salons_table

pre exists[i: inset dom(single_salon_services)] | single_salon_services(i).id = salon_id
post not exists[i: inset dom(single_salon_services)] | i = deleted_service and single_salon_services(i).id = salon_id
```

```
578                => access_token = validtoken
579                  and len(access_token) <> 0
580                    and access_token <> nil
581                      and single_salon_services = override(~single_salon_services, {deleted_service-->salon})
582                        and deleted_service.id = service_id
583                          and exists[i: inset dom(salons_table)] | i.id = salon_id and salons_table(i) = salonuser
584                            and salonuser.role = <SALONOWNER>
585                              and response_message = "Http 204"
586           or
587              error_message = "Http 401, Permission Denied"
588                and access_token <> validtoken
589                  and len(access_token) = 0
590                    and access_token = nil
591                      and exists[i: inset dom(salons_table)] | i.id = salon_id and salons_table(i) = salonuser
592                        and salonuser.role = <SALONOWNER>
593 end_process;
594
595 process GetSalonCustomers(validtoken: token, access_token: token, salon_id: string) customers: MapSalonCustomerToSalon,
596     response_status: string | error_message: string
597 /* API resource path: /salons/{salon_id}/customers */
598
599 ext rd salon_customers_table
600
601 pre exists[i: inset rng(salon_customers_table)] | i.id = salon_id
602 post forall[x: inset dom(salon_customers_table)] | x inset dom(customers) and customers(x).id = salon_id and subset(dom(customers),
603     dom(salon_customers_table)) <=> true
604                => exists[i: inset rng(salon_customers_table)] | i.id = salon_id
605                  and access_token = validtoken
606                    and len(access_token) <> 0
607                      and access_token <> nil
608                        and response_status = "Http 200 Ok"
609            or
610              error_message = "Http 401, Permission Denied"
611                and access_token <> validtoken
612                  and len(access_token) = 0
613                    and access_token = nil
614                      and exists[i: inset rng(salon_customers_table)] | i.id = salon_id
615
616 end_process;
617
618 process AddSalonCustomer(validtoken: token, access_token: token, salon: SalonData, salonuser, saloncustomer: SalonUser) added_customer: SalonUser,
619     response_status: string | error_message: string
620 /* API resource path: /salons/{salon_id}/customers */
621 ext wr salon_customers_table
622 ext rd salons_table
623 ext rd salon_users_table
624
625 pre not exists[i: inset dom(salon_customers_table)] | i = saloncustomer and i.role = <SALONCUSTOMER>
626                and exists[j: inds(salon_users_table)] | j.role = <SALONCUSTOMER>
```

```
post salon_customers_table = override(~salon_customers_table, {saloncustomer-->salon}) and added_customer inset dom(salon_customers_table)
                  => access_token = validtoken
                    and len(access_token) <> 0
                      and access_token <> nil
                        and exists[i: inset dom(salon_customers_table)] | i = saloncustomer and i.role = <SALONCUSTOMER> and salon_customers_table(i) = salon
                          and salons_table(salon) = salonuser
                            and salonuser.role = <SALONOWNER>
                              and response_message = "Http 200 Ok"
              or
                error_message = "Http 401, Permission Denied"
                  and access_token <> validtoken
                    and len(access_token) = 0
                      and access_token = nil
                        and exists[i: inds(salon_users_table)] | salon_users_table(i) = salonuser and salon_users_table(i).role = <SALONOWNER>
                          and not exists[j: inset dom(salon_customers_table)] | j = saloncustomer and j.role = <SALONCUSTOMER>


end_process;

process GetSalonCustomer(validtoken: token, access_token: token, salon_id: string, customer_id: string) saloncustomer: SalonUser,
    response_status: string | error_message: string
/* API resource path: /salons/{salon_id}/customers/{customer_id} */
ext rd salon_customers_table

post exists![i: inset dom(salon_customers_table)] | i = saloncustomer | salon_customers_table(i).id = salon_id
              => access_token = validtoken
                and len(access_token) <> 0
                    and access_token <> nil
                      and saloncustomer inset dom(salon_customers_table)
                        and saloncustomer.id = customer_id
                          and saloncustomer.role = <SALONCUSTOMER>
                            and response_status = "Http 200 Ok"
            or
            error_message = "Http 401, Permission Denied"
              and access_token <> validtoken
                and len(access_token) = 0
                  and access_token = nil
                    and exists![i: inset dom(salon_customers_table)] | i.id = customer_id and saloncustomer.id = customer_id
                        and salon_customers_table(i).id = salon_id

end_process;

process DeleteSalonCustomer(validtoken: token, access_token: token, salon: SalonData, customer_id: string) deleted_customer: SalonUser,
    response_status: string | error_message: string

ext wr salon_customers_table

pre exists[i: inset dom(salon_customers_table)] | i.id = customer_id
post salon_customers_table = override(~salon_customers_table, {deleted_customer-->salon })
```

```
676             => access_token = validtoken
677               and len(access_token) <> 0
678                   and access_token <> nil
679                     and exists[i: inset dom(~salon_customers_table)] | i.id = customer_id and salon_customers_table(i) = salon
680                       and deleted_customer.id = customer_id
681                         and salon_customers_table(deleted_customer) = salon
682                           and deleted_customer.role = <SALONCUSTOMER>
683                             and response_status = "Http 204"
684           or
685             error_message ="Http 401, Permission Denied"
686                           and access_token <> validtoken
687                             and len(access_token) = 0
688                               and access_token = nil
689                                 and exists[i: inset dom(~salon_customers_table)] | i.id = customer_id and salon_customers_table(i) = salon
690
691 end_process;
692
693
694 process GetSalonBookings(validtoken: token, access_token: token, user: SalonUser) bookings: MapSalonBookingsToCustomer,
695     response_status: string | error_message: string
696 /* get salon bookings for a salon customer */
697
698 ext rd salon_bookings_table
699
700 pre exists[i: inset dom(salon_bookings_table)] | salon_bookings_table(i) = user
701 post forall [x: inset dom(bookings)] | x inset dom(salon_bookings_table) and bookings(x) = user and subset(dom(bookings), dom(salon_bookings_table)) <=> true
702             => access_token = validtoken
703             and len(access_token) <> 0
704               and access_token <> nil
705                 exists[i: inset dom(salon_bookings_table)] | salon_bookings_table(i) = user
706                   and user.role = <SALONCUSTOMER>
707                     and response_status = "Http 200 Ok"
708           or
709           error_message = "Http 401, Permission Denied"
710             and access_token <> validtoken
711               and len(access_token) = 0
712                 and access_token = nil
713                   and user.role = <SALONCUSTOMER>
714
715 end_process;
716
717 process GetSalonBooking(validtoken: token, access_token: token, user: SalonUser, booking_id: string) booking: SalonBookingData,
718     response_status: string | error_message:string
719 /* Get single booking for a customer */
720 ext rd salon_bookings_table
721
722 pre exists[i: inset dom(salon_bookings_table)] | i.id = booking_id and salon_bookings_table(i) = user
723 post exists![i: inset dom(salon_bookings_table)] | i.id = booking_id and booking.id = booking_id | salon_bookings_table(i) = user
724                 => access_token = validtoken
```

```
                and len(access_token) <> 0
                  and access_token <> nil
                    and booking inset dom(salon_bookings_table)
                      and salon_bookings_table(booking) = user
                        and salon_bookings_table(boooking).role = <SALONCUSTOMER>
                          and response_status = "Http 200 Ok"
            or
              error_message = "Http 401, Permission Denied"
                and access_token <> validtoken
                  and len(access_token) = 0
                    and access_token = nil
                      and exists![i: inset dom(salon_bookings_table)] | i.id = booking_id | salon_bookings_table(i) = user and user.role = <SALONCUSTOMER>

end_process;

process DeleteSalonBooking(validtoken: token, access_token: token, user: SalonUser, booking_id: string) deleted_booking: SalonBookingData,
    response_status: string |  error_message: string
/* API resource path: /salons/{user_id}/bookings/{booking_id}
Delete booking of a customer */

ext wr salon_bookings_table

pre exists[i: inset dom(salon_bookings_table)] | i.id = booking_id | salon_bookings_table(i) = user
post salon_bookings_table = override(~salon_bookings_table, {deleted_booking-->user}) and user.role = <SALONOWNER>
            => access_token = validtoken
              and len(access_token) <> 0
                  and access_token <> nil
                    and exists![i: inset dom(salon_bookings_table)] | i.id = booking_id and salon_bookings_table(i) = user
                      and salon_bookings_table(deleted_booking) = user
                        and user.role = <SALONCUSTOMER>
                          and response_status = "Http 204"
            or
            error_message = "Http 401, Permission Denied"
              and access_token <> validtoken
                and len(access_token) = 0
                  and access_token = nil
                    and exists![i: inset dom(salon_bookings_table)] | i.id = booking_id and salon_bookings_table(i) = user and user.role = <SALONOWNER>
end_process;
```

Listing A.1: Salon Online Booking System Case Study

Bibliography

[1] Mike Amundsen Leonard Richardson and Sam Ruby. *RESTful Web APIs*. First Edition. O'reilly Media Inc. 1005 Gravenstein Highway North Sebastopol CA 95472, Sept. 2013. ISBN: 978-1-4493-5806-8. URL: https://www.oreilly.com/catalog/errata.csp?isbn=9781449358068.

[2] P. Siriwardena. *Advanced API Security: OAuth 2.0 and Beyond*. Apress, 2019. ISBN: 9781484220504. URL: https://books.google.co.jp/books?id=47jEDwAAQBAJ.

[3] R.T Fielding. *Architectural Styles and the Design of Network-based Software Architectures - Ph.D. dissertation*. University of California, Irvine, 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

[4] P. Siriwardena and N. Dias. *Microservices Security in Action*. In Action. Manning Publications, 2020. ISBN: 9781617295959. URL: https://books.google.co.jp/books?id=f_f1DwAAQBAJ.

[5] I. Ristic. *Bulletproof SSL and TLS: Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications*. Computers / Security. Feisty Duck, 2013. ISBN: 9781907117046. URL: https://books.google.co.jp/books?id=fQOLBAAAQBAJ.

[6] *HTTP headers - HTTP — MDN*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers. (Accessed on 04/25/2022).

[7] *JSON*. https://www.json.org/json-en.html. (Accessed on 04/07/2022).

[8] *OWASP Top Ten Web Application Security Risks — OWASP*. https://owasp.org/www-project-top-ten/. (Accessed on April.25.2022).

[9] Neil Madden. *API Security in Action*. Manning Publications Co., Nov. 2020. ISBN: 978-1-61729-602-4.

[10] *sqlmap: automatic SQL injection and database takeover tool*. https://sqlmap.org/. (Accessed on 05/12/2022).

[11] Bertrand Meyer. *Eiffel: The Language*. USA: Prentice-Hall, Inc., 1992. ISBN: 0132479257.

[12] A. Hall and D. Isaac. "Formal methods in a real air traffic control project". In: *IEE Colloquium on Software in Air Traffic Control Systems - The Future*. 1992, pp. 7/1–7/4.

[13] Mohammad Reza Nami and Abbas Malekpour. "Formal specification of a particular banking domain with RAISE specification language". In: *2008 IEEE Symposium on Computers and Communications*. 2008, pp. 695–699. DOI: 10.1109/ISCC.2008.4625588.

[14] S.P. Miller and M. Srivas. "Formal verification of the AAMP5 microprocessor: a case study in the industrial use of formal methods". In: *Proceedings of 1995 IEEE Workshop on Industrial-Strength Formal Specification Techniques*. 1995, pp. 2–16. DOI: 10.1109/WIFT.1995.515475.

[15] Shaoying Liu. *Formal Engineering for Industrial Software Development Using the SOFL Method*. Springer-Verlag Berlin Heidelberg New York, 2004. ISBN: 3-540-20602-7.

[16] Atif Mashkoor and Jean-Pierre Jacquot. "Stepwise Validation of Formal Specifications". In: *2011 18th Asia-Pacific Software Engineering Conference*. 2011, pp. 57–64. DOI: 10.1109/APSEC.2011.48.

[17] D. Hazel, P. Strooper, and O. Traynor. "Requirements engineering and verification using specification animation". In: *Proceedings 13th IEEE International Conference on Automated Software Engineering (Cat. No.98EX239)*. 1998, pp. 302–305. DOI: 10.1109/ASE.1998.732685.

[18] I. Morrey et al. "Use of a specification construction and animation tool to teach formal methods". In: *Proceedings of 1993 IEEE 17th International Computer Software and Applications Conference COMPSAC '93*. 1993, pp. 327–333. DOI: 10.1109/CMPSAC.1993.404236.

[19] E. Kazmierczak, M. Winikoff, and P. Dart. "Verifying model oriented specifications through animation". In: *Proceedings 1998 Asia Pacific Software Engineering Conference (Cat. No.98EX240)*. 1998, pp. 254–261. DOI: 10.1109/APSEC.1998.733727.

[20] Charlotte Seidner and Olivier H. Roux. "Formal Methods for Systems Engineering Behavior Models". In: *IEEE Transactions on Industrial Informatics* 4.4 (2008), pp. 280–291. DOI: 10.1109/TII.2008.2008998.

[21] Shaoying Liu et al. "SOFL: a formal engineering methodology for industrial applications". In: *IEEE Transactions on Software Engineering* 24.1 (1998), pp. 24–45. DOI: 10.1109/32.663996.

[22] Shaoying Liu. "SOFL: a formal engineering methodology for industrial applications". In: *Proceedings of ISRE '97: 3rd IEEE International Symposium on Requirements Engineering*. 1997, pp. 41–. DOI: 10.1109/ISRE.1997.566840.

[23] I. Tarandach and M.J. Coles. *Threat Modeling*. O'Reilly Media, 2020. ISBN: 9781492056508. URL: https://books.google.co.jp/books?id=ZcQIEAAAQBAJ.

[24] A. Hathaway and T. Hathaway. *Data Flow Diagrams - Simply Put!: Process Modeling Techniques for Requirements Elicitation and Workflow Analysis*. Advanced Business Analysis Topics. CreateSpace Independent Publishing Platform, 2016, p. 71. ISBN: 9781535110136. URL: https://books.google.co.jp/books?id=8SipzgEACAAJ.

[25] R.E. Giachetti. *Design of Enterprise Systems: Theory, Architecture, and Methods*. CRC Press, 2016. ISBN: 9781439882894. URL: https://books.google.co.jp/books?id=PbTMBQAAQBAJ.

[26] Matthias Biehl. *API Architecture: The Big Picture for Building APIs*. Vol. 2. API University Series, 2015. ISBN: 978-1-5086-7664-5.

[27] Kordy et al. "Attack-defense trees". In: *Journal of Logic and Computation*. 24 (2014). DOI: 10.1093/logcom/exs029.

[28] Pethuru Raj Harihara Subramanian. *Hands-On RESTful API Design Patterns and Best Practices*. Packt Publishing, Jan. 2019. ISBN: 978-1-78899-266-4.

[29] Kirsten L Hunter. *Irresistible APIs- Designing Web APIs that Developers will Love*. Manning Publications Co., 2017. ISBN: 978-1-61729-255-2.

[30] Cliff Jones. "Systematic Software Development Using VDM". In: Jan. 1990.

[31] E. Evans, M. Fowler, and E.J. Evans. *Domain-driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004. ISBN: 9780321125217. URL: https://books.google.co.jp/books?id=xColAAPGubgC.

[32] D. Sawano, D.B. Johnsson, and D. Deogun. *Secure by Design*. Manning, 2019. ISBN: 9781638352310. URL: https://books.google.co.jp/books?id=mzozEAAAQBAJ.

[33] Dave Steinberg, ed. *EMF: Eclipse Modeling Framework*. en. 2nd ed., Rev. and updated. The eclipse series. OCLC: ocn182662768. Upper Saddle River, NJ: Addison-Wesley, 2009. ISBN: 978-0-321-33188-5.

[34] *raml-spec/raml-10.md at master · raml-org/raml-spec · GitHub*. https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/#markup-language. (Accessed on 04/07/2022).

[35] *Daring Fireball: Markdown*. https://daringfireball.net/projects/markdown/. (Accessed on 04/07/2022).

[36] *The Official YAML Web Site*. https://yaml.org/. (Accessed on 04/07/2022).

[37] *OpenAPI Specification - Version 3.0.3 — Swagger*. https://swagger.io/specification/. (Accessed on 04/07/2022).

[38] *SwaggerHub — API Design and Documentation with OpenAPI*. https://swagger.io/tools/swaggerhub/. (Accessed on 04/07/2022).

[39] H. Mouratidis. *Software Engineering for Secure Systems: Industrial and Research Perspectives: Industrial and Research Perspectives*. Premier reference source. Information Science Reference, 2010. ISBN: 9781615208388. URL: https://books.google.co.jp/books?id=RBTK7QfYquoC.

[40] K. S. HOO. "Tangible ROI through Secure Software Engineering". In: *Security Business Quarterly* (2001). URL: https://ci.nii.ac.jp/naid/10019397603/en/.

[41] Busalire Emeka and Shaoying Liu. "A Formal Technique for Concurrent Generation of Software's Functional and Security Requirements in SOFL Specifications". In: *Structured Object-Oriented Formal Language and Method*. Ed. by Huaikou Miao et al. Cham: Springer International Publishing, 2020, pp. 13–28. ISBN: 978-3-030-41418-4.

[42] Busalire Emeka, Soichiro Hidaka, and Shaoying Liu. "A Formal Approach to Secure Design of RESTful Web APIs Using SOFL". In: *Structured Object-Oriented Formal Language and Method*. Ed. by Jinyun Xue et al. Cham: Springer International Publishing, 2021, pp. 105–125. ISBN: 978-3-030-77474-5.

[43] *The Common Criteria Portal - Security Functional Requirements*. https://www.commoncriteriaportal.org/cc/. (Accessed on 02/26/2023).

[44] *AICPA Privacy Management Framework*. https://us.aicpa.org/interestareas/informationtechnology/privacy-management-framework. (Accessed on 02/26/2023).

[45] Nancy R. Mead, Dan Shoemaker, and Jeff Ingalsbe. "Teaching Security Requirements Engineering Using SQUARE". In: *2009 Fourth International Workshop on Requirements Engineering Education and Training*. 2009, pp. 20–27. DOI: 10.1109/REET.2009.12.

[46] Nancy R. Mead and Ted Stehney. "Security Quality Requirements Engineering (SQUARE) Methodology". In: *SIGSOFT Softw. Eng. Notes* 30.4 (2005), 1–7. ISSN: 0163-5948. DOI: 10.1145/1082983.1083214. URL: https://doi.org/10.1145/1082983.1083214.

[47] M.S. Merkow and L. Raghavan. *Secure and Resilient Software: Requirements, Test Cases, and Testing Methods*. CRC Press, 2011. ISBN: 9781439866221. URL: https://books.google.co.jp/books?id=0PTRBQAAQBAJ.

[48] Busalire Emeka, Soichiro Hidaka, and Shaoying Liu. "A Formal Approach to Secure Design of RESTful Web APIs Using SOFL". In: *Structured Object-Oriented Formal Language and Method*. Ed. by Jinyun Xue et al. Cham: Springer International Publishing, 2021, pp. 105–125. ISBN: 978-3-030-77474-5.

[49] Li Jiang, Hao Chen, and Fei Deng. "A Security Evaluation Method Based on STRIDE Model for Web Service". In: *2010 2nd International Workshop on Intelligent Systems and Applications*. 2010, pp. 1–5. DOI: 10.1109/IWISA.2010.5473445.

[50] Shaoying Liu et al. "SOFL: a formal engineering methodology for industrial applications". en. In: *IEEE Transactions on Software Engineering* 24.1 (Jan. 1998), pp. 24–45. ISSN: 00985589. DOI: 10.1109/32.663996. URL: http://ieeexplore.ieee.org/document/663996/ (visited on 03/24/2020).

[51] Jan Jurjens. "UMLsec: Extending UML for Secure Systems Development". In: *UML 2002 — The Unified Modeling Language*. Ed. by Jean-Marc Jezequel, Heinrich Hussmann, and Stephen Cook. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 412–425. ISBN: 978-3-540-45800-5.

[52] Haralambos Mouratidis, Paolo Giorgini, and Gordon Manson. "An Ontology for Modelling Security: The Tropos Approach". In: *Knowledge-Based Intelligent Information and Engineering Systems*. Ed. by Vasile Palade, Robert J. Howlett, and Lakhmi Jain. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 1387–1394. ISBN: 978-3-540-45224-9.

[53] Nicolas Mayer et al. "Towards a Risk-Based Security Requirements Engineering Framework". In: *In Proc. of Requirements Engineering for Software Quality'05*. 2005, pp. 89 –104.

[54] A. van Lamsweerde and E. Letier. "Handling obstacles in goal-oriented requirements engineering". In: *IEEE Transactions on Software Engineering* 26.10 (2000), pp. 978–1005. DOI: 10.1109/32.879820.

[55] Renaud De Landtsheer and Axel van Lamsweerde. "Reasoning about Confidentiality at Requirements Engineering Time". In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-13. Lisbon, Portugal: Association for Computing Machinery, 2005, 41–49. ISBN: 1595930140. DOI: 10.1145/1081706.1081715. URL: https://doi.org/10.1145/1081706.1081715.

[56] Axel Lamsweerde. "Engineering Multi-View Models for Model-Driven Engineering". In: Birmingham, United Kingdom, July 2013, pp. 3–3. DOI: 10.1109/TASE.2013.8.

[57] Jon Whittle, Duminda Wijesekera, and Mark Hartong. "Executable misuse cases for modeling security concerns". In: Leipzig, Germany, June 2008, pp. 121 –130. DOI: 10.1145/1368088.1368106.

[58] Daniel Fett, Pedram Hosseyni, and Ralf Küsters. "An Extensive Formal Security Analysis of the OpenID Financial-Grade API". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 453–471. DOI: 10.1109/SP.2019.00067.

[59] Jacek Kopecký, Karthik Gomadam, and Tomas Vitvar. "hRESTS: An HTML Microformat for Describing RESTful Web Services". en. In: *2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*. Sydney, Australia: IEEE, Dec. 2008, pp. 619–625. ISBN: 978-0-7695-3496-1. DOI: 10.1109/WIIAT.2008.379. URL: http://ieeexplore.ieee.org/document/4740521/ (visited on 10/18/2021).

[60] Sultan S. Alqahtani, Ellis E. Eghan, and Juergen Rilling. "Recovering Semantic Traceability Links between APIs and Security Vulnerabilities: An Ontological Modeling Approach". en. In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Tokyo, Japan: IEEE, Mar. 2017, pp. 80–91. ISBN: 978-1-5090-6031-3. DOI: 10.1109/ICST.2017.15. URL: http://ieeexplore.ieee.org/document/7927965/ (visited on 03/27/2020).

[61] Uri Klein and Kedar S Namjoshi. "Formalization and automated verification of RESTful behavior". In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 541–556.

[62] Irum Rauf, Inna Vistbakka, and Elena Troubitsyna. "Formal Verification of Stateful Services with REST APIs Using Event-B". In: *2018 IEEE International Conference on Web Services (ICWS)*. 2018, pp. 131–138. DOI: 10.1109/ICWS.2018.00024.

[63] K. Hunter. *Irresistible APIs: Designing web APIs that developers will love*. Manning, 2016. ISBN: 9781638353447. URL: https://books.google.co.jp/books?id=dTkzEAAAQBAJ.

[64] M. Brambilla et al. *Model-Driven Software Engineering in Practice: Second Edition*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2017. ISBN: 9781627059886. URL: https://books.google.co.jp/books?id=bXSbDgAAQBAJ.

[65] Juan Cadavid, Benoit Combemale, and Benoit Baudry. *Ten years of Meta-Object Facility: an Analysis of Metamodeling Practices*. Research Report RR-7882. INRIA, Feb. 2012. URL: https://hal.inria.fr/hal-00670652.

[66] Jean Bézivin et al. "Modeling in the Large and Modeling in the Small". In: *European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004* 3599 (Jan. 2004), pp. 33–46. DOI: 10.1007/11538097_3.

[67] Jean Bézivin. "On the unification power of models". In: *Software & Systems Modeling* 4.2 (May 2005), pp. 171–188. ISSN: 1619-1374. DOI: 10.1007/s10270-005-0079-0. URL: https://doi.org/10.1007/s10270-005-0079-0.

[68] *Meta Object Facility (MOF) Core Specification*. Object Management Group, Jan. 2006.

[69] *org.eclipse.emf.ecore (EMF Documentation)*. URL: https://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html (visited on 02/28/2022).

[70] *Epsilon book*. URL: https://www.eclipse.org/epsilon/doc/book/.

[71] Louis Rose et al. "The Epsilon Generation Language". In: June 2008, pp. 1–16. ISBN: 978-3-540-69095-5. DOI: 10.1007/978-3-540-69100-6_1.

[72] Mark Nottingham. *Web Linking*. Tech. rep. 5988. Oct. 2010. 23 pp. DOI: 10.17487/RFC5988. URL: https://rfc-editor.org/rfc/rfc5988.txt.

[73] S. Liu. "A rigorous approach to reviewing formal specifications". In: *27th Annual NASA Goddard-/IEEE Software Engineering Workshop, 2002. Proceedings.* 2002, pp. 75–81. DOI: 10.1109/SEW.2002.1199452.

[74] Shaoying Liu. "Verifying formal specifications using fault tree analysis". In: *Proceedings International Symposium on Principles of Software Evolution*. 2000, pp. 272–281. DOI: 10.1109/ISPSE.2000.913248.

[75] M. Silva. "Introducing Petri nets". In: *Practice of Petri Nets in Manufacturing*. Dordrecht: Springer Netherlands, 1993, pp. 1–62. ISBN: 978-94-011-6955-4. DOI: 10.1007/978-94-011-6955-4_1. URL: https://doi.org/10.1007/978-94-011-6955-4_1.

[76] C.B. Jones. *Systematic Software Development Using VDM*. 2nd Edition. Prentice Hall, 1990.

[77] C. Ho-Stuart and Shaoying Liu. "A formal operational semantics for SOFL". In: *Proceedings of Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference*. 1997, pp. 52–61. DOI: 10.1109/APSEC.1997.640161.

[78] *Common Vulnerability Enumeration - CVE*. https://cve.mitre.org/. (Accessed on Dec.21.2022).

[79] *raml-org/raml-java-parser: (deprecated) A RAML parser based on SnakeYAML written in Java*. https://github.com/raml-org/raml-java-parser. (Accessed on 08/09/2022).

[80] *X-XSS-Protection - HTTP — MDN*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-XSS-Protection. (Accessed on 02/19/2023).

[81] Shaoying Liu and Shin Nakajima. "A Decompositional Approach to Automatic Test Case Generation Based on Formal Specifications". In: *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*. 2010, pp. 147–155. DOI: 10.1109/SSIRI.2010.11.

[82] Shaoying Liu et al. "An Automated Approach to Specification-Based Program Inspection". In: *Formal Methods and Software Engineering*. Ed. by Kung-Kiu Lau and Richard Banach. Berlin, Heidelberg: Springer Berlin, Heidelberg, 2005, pp. 421–434. ISBN: 978-3-540-32250-4.

[83] *Acceleo/User Guide - Eclipsepedia*. URL: https://wiki.eclipse.org/Acceleo/User_Guide (visited on 02/27/2022).

[84] *Epsilon*. URL: https://www.eclipse.org/epsilon/ (visited on 02/27/2022).

[85] Marzieh Ghorbani, Mohammadreza Sharbaf, and Bahman Zamani. "Incremental Model Transformation with Epsilon in Model-Driven Engineering". In: *Acta Informatica Pragensia* 11 (Apr. 2022), pp. 179–204. DOI: 10.18267/j.aip.179.

[86] Mo Li and Shaoying Liu. "Integrating Animation-Based Inspection Into Formal Design Specification Construction for Reliable Software Systems". In: *IEEE Transactions on Reliability* 65.1 (2016), pp. 88–106. DOI: 10.1109/TR.2015.2456853.

[87] *The web framework for perfectionists with deadlines — Django*. https://www.djangoproject.com/. (Accessed on 04/09/2022).

[88] *Home - Django REST framework*. https://www.django-rest-framework.org/. (Accessed on 04/09/2022).

[89] *Python Release Python 3.5.0 — Python.org*. https://www.python.org/downloads/release/python-350/. (Accessed on 04/09/2022).

[90] *PostgreSQL: PostgreSQL 14.2, 13.6, 12.10, 11.15, and 10.20 Released!* https://www.postgresql.org/about/news/postgresql-142-136-1210-1115-and-1020-released-2402/. (Accessed on 04/09/2022).

[91] *draw.io*. https://drawio-app.com/. (Accessed on 04/09/2022).

[92] *Epsilon*. https://www.eclipse.org/epsilon/. (Accessed on 04/09/2022).

[93] *Java Downloads — Oracle*. https://www.oracle.com/java/technologies/downloads/. (Accessed on 04/09/2022).

[94] *ADTool*. https://satoss.uni.lu/members/piotr/adtool/. (Accessed on 03/07/2022).

[95] *Getting Started - Epsilon*. https://www.eclipse.org/epsilon/getting-started/. (Accessed on 03/07/2022).

[96] A. Hoffman. *Web Application Security: Exploitation and Countermeasures for Modern Web Applications*. O'Reilly Media, 2020. ISBN: 9781492053088. URL: https://books.google.co.jp/books?id=3R3UDwAAQBAJ.

[97] M. Shema. *Seven Deadliest Web Application Attacks*. ITPro collection. Elsevier Science, 2010. ISBN: 9781597495448. URL: https://books.google.co.jp/books?id=Kb08r1Yy0rYC.