國立臺灣大學電機資訊學院資訊網路與多媒體研究所

碩士論文

Graduate Institute of Networking and Multimedia

College of Electrical Engineering and Computer Science

National Taiwan University

Master's Thesis

基於逆向工程技術之前端 Angular 程式碼至使用案例模型建構

Reverse Engineering Frontend Angular Code to Create a Use Case Model

陳詠君

Yung-Chun Chen

指導教授: 李允中 博士

Advisor: Jonathan Lee Ph.D.

中華民國 114 年 7 月

July, 2025

doi:10.6342/NTU202503567

國立臺灣大學碩士學位論文
口試委員會審定書
MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

基於逆向工程技術之前端 Angular 程式碼至使用案例模型建構

Reverse Engineering Frontend Angular Code to Create a Use Case Model

本論文係 陳詠君 （學號 R12944020）在國立臺灣大學資訊網路與多媒體研究所完成之碩士學位論文，於民國 114 年 7 月 31 日承下列考試委員審查通過及口試及格，特此證明。

The undersigned, appointed by the Graduate Institute of Networking and Multimedia on 31 July 2025 have examined a Master's Thesis entitled above presented by CHEN, YUNG-CHUN (student ID: R12944020) candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

孫宏民	劉志俊	陳錫民
（指導教授 Advisor）		
馬尚彬	李文正	

系（所）主管 Director: 鄭卜壬

Acknowledgements

首先，我要感謝李允中教授這兩年來的悉心指導，教導我遇到問題時，先把問題拆解、再逐一分析問題、尋找解決方法，並且在我的研究過程中給我指引。

我也要感謝我的家人，在我繁忙的研究所生活中，雖然關心不常掛在嘴邊，卻始終默默陪伴在我身旁，讓我能無後顧之憂地全力衝刺碩士學業。並且，我要感謝臺灣大學軟體工程實驗室的成員: 陳可瀚、康甜甜、吳東鴻、葉晟育、鄭佳渝、謝承恩、許耀允、朱立揚、錢怡君以及所有實驗室的學長姐、學弟妹的幫忙，協助我的研究。

doi:10.6342/NTU202503567

摘要

現代的 Angular 專案普遍採用元件化設計，導致程式碼量龐大且結構複雜，使得僅透過人工閱讀程式碼來理解系統功能與使用者需求變得極具挑戰性。

為了解決此問題，我們提出一套逆向工程方法，旨在從前端程式碼中自動化建構使用案例模型。此方法透過將 Angular 程式碼解析為抽象語法樹，進一步分析元件之間的相依性與互動邏輯，以挖掘出潛藏於程式碼中的系統行為與使用情境。

最終所產生的使用案例模型可協助開發人員、測試人員、分析師或專案使用者更有效率地掌握系統意圖，並促進系統維護、需求追蹤與功能擴充。此方法不僅提升了理解系統的效率，也為大型前端系統的分析提供了一種新方向。

關鍵字： Angular、逆向工程、使用案例模型、抽象語法樹、前端分析

doi:10.6342/NTU202503567

Abstract

Modern Angular projects typically adopt a component-based architecture, resulting in large and complex codebases. This makes it highly challenging to understand system functionality and user requirements merely by reading the source code.

To address this issue, we propose a reverse engineering approach that automatically constructs use case models from frontend code. By parsing Angular code into an Abstract Syntax Tree and analyzing component dependencies and interaction logic, our method uncovers the embedded system behaviors and usage scenarios hidden in the code.

The resulting use case models help developers, testers, analysts, or project users efficiently grasp the system's intended behavior, while also facilitating system maintenance, requirement tracking, and functional expansion. This approach not only improves the efficiency of understanding the system but also offers a new direction for the analysis of large-scale frontend systems.

Keywords: Angular, Reverse Engineering, Use Case Model, Abstract Syntax Tree, Fron-

doi:10.6342/NTU202503567

tend Analysis

doi:10.6342/NTU202503567

Contents

doi:10.6342/NTU202503567

doi:10.6342/NTU202503567

doi:10.6342/NTU202503567

List of Figures

doi:10.6342/NTU202503567

doi:10.6342/NTU202503567

List of Tables

doi:10.6342/NTU202503567

Chapter 1 Introduction

In modern web application development, frontend architectures have become increasingly modular and complex. Especially with the widespread adoption of the Angular framework, developers tend to organize entire systems using numerous components and modules. While this architecture enhances maintainability and reusability, it also results in system logic being distributed across multiple files and layers. Consequently, users often find it difficult to fully comprehend the system's actual behavior and operational flow. Moreover, many real-world projects lack formal and complete use case models or requirement specification documents. Even if such documents exist, they often become outdated and inconsistent with the actual system behavior due to frequent system updates. This situation poses significant challenges for both users and developers in understanding system functionality. It becomes particularly problematic during feature expansion, bug fixing, or system maintenance, as misinterpreting the logic can lead to incorrect operations, ultimately affecting overall system quality and development efficiency.

This study proposes a reverse engineering approach to analyze Angular frontend code and automatically construct the corresponding use case model. The core of this approach involves converting Angular code into an Abstract Syntax Tree, and then identifying component dependencies, event trigger points, HTTP request flows, and their corresponding user behaviors through node parsing and structural analysis. Based on this information, a

doi:10.6342/NTU202503567

use case model for the project is derived.

The organization of this thesis is as follows: Chapter 2 introduces the related work; Chapter 3 presents the overall system architecture of this study; Chapter 4 discusses the relationships among programming languages used in Angular frontend code; Chapter 5 explains how AST nodes are extracted, including the design of the entity-relationship diagram and class diagram for visiting AST nodes and storing them in the database; Chapter 6 describes the automatic construction of the use case model based on frontend dependencies; Chapter 7 provides the conclusion; and Chapter 8 outlines directions for future work.

doi:10.6342/NTU202503567

Chapter 2 Related Work

2.1 Recovering Use Case Models from Source Code

Previous studies have explored how to derive use case models from source code. Miranda et al. focused on GUI-based software systems and proposed a reverse engineering technique for constructing use case models [1]. They first extracted and filtered static information from the system, particularly GUI components, and then evaluated the importance of methods or functions using a set of metrics. Subsequently, they performed summarization and clustering to group related software artifacts, and mapped these clusters to a UML use case model. This approach provides detailed and representative use cases, which help engineers better understand system behavior.

2.2 Reverse Engineering

Reverse engineering techniques [2][3][4][5] play a pivotal role in the field of software development. Their primary purpose is to assist developers in gaining a deeper understanding of the existing software system architecture and operational logic. By analyzing source code, system behavior, and interactions among modules, reverse engineering can effectively recover the original design concepts, thereby supporting system refactoring and

doi:10.6342/NTU202503567

redevelopment. This is especially valuable in situations where complete design documentation is lacking or the original development team is no longer available for maintenance. In such cases, reverse engineering becomes a crucial means to understand legacy systems, enhance maintainability, and extend system lifespan.

Chikofsky and Cross II [5] define reverse engineering as "a process of identifying the components of a system and their interrelationships, analyzing the system, and creating a higher-level abstraction of the system." They emphasize that reverse engineering is an analytical process that does not involve modifying the system or creating a new one. This definition remains one of the most widely accepted and representative descriptions of reverse engineering to date.

Through reverse engineering techniques, we can gain deep insights into the actual operation of frontend systems, with the core being in-depth analysis of source code. Using API call analysis, event handler analysis, user interaction analysis, and system behavior analysis derived from the source code, we can gradually reconstruct the real logic and processes executed during system operation. This kind of static analysis technique can effectively capture the mapping between user actions and system responses, thereby constructing use case models that closely reflect real operational scenarios.

For this study, reverse engineering serves as an effective technical approach to help users quickly understand the structure and behavior of a system. Even in the absence of complete documentation or in the case of large-scale systems, it enables the automated extraction of critical operational flows and logic through code analysis. This is particularly helpful for modeling use scenarios in frontend applications, as it not only improves the accuracy and consistency of use case models but also helps users gain a more comprehensive

doi:10.6342/NTU202503567

understanding of how the system responds to various interactive situations.

2.3 AngularJS and Angular

Angular is a frontend framework developed by Google, designed for building dynamic and high-performance web applications. It has two major versions: AngularJS (version 1.x) [6] and Angular (version 2 and later) [7].

2.3.1 AngularJS

AngularJS (version 1.x) is a JavaScript-based frontend framework developed by Google, used for building dynamic web applications, particularly well-suited for developing Single Page Applications. It combines JavaScript and HTML, and through data binding and template syntax, allows developers to more efficiently construct interactive user interfaces. It adopts the MVC architecture, helping developers organize and manage code more effectively.

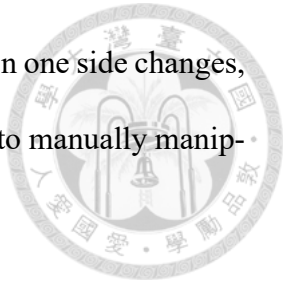
One of the standout features of AngularJS is its support for two-way data binding and dependency injection. These mechanisms greatly simplify the development process and enhance user experience, which led to its rapid adoption and made it one of the most popular frontend frameworks at the time. AngularJS has the following key features:

MVC Architecture

The application is divided into Model, View, and Controller, making the code structure clear, maintainable, and suitable for collaborative development.

Two-Way Data Binding

doi:10.6342/NTU202503567



Data between the model and view is automatically synchronized. When one side changes, the other reflects it in real-time. This reduces the need for developers to manually manipulate the DOM, improving development efficiency.

Dependency Injection

Dependency injection is a design pattern that implements inversion of control. With this approach, components do not need to create their own dependencies; instead, the framework injects the required dependencies at runtime. This reduces coupling between modules and improves modularity, reusability, and testability of the code.

Directives

AngularJS extends HTML syntax, allowing developers to define custom HTML tags and attributes to encapsulate behavior and logic. These directives are used to manipulate the DOM or display data, enhancing code structure and readability.

Single Page Applications (SPA)

AngularJS enables the creation of applications that only need to be loaded once. By using routing and data binding techniques, the frontend dynamically updates content, improving page responsiveness and user experience.

With its innovative design concepts and rich features, AngularJS brought a revolutionary shift to frontend development at the time. It simplified data processing and page updates, making the development of modern web applications more efficient and modular. It remains one of the most representative technologies in the history of frontend frameworks.

doi:10.6342/NTU202503567

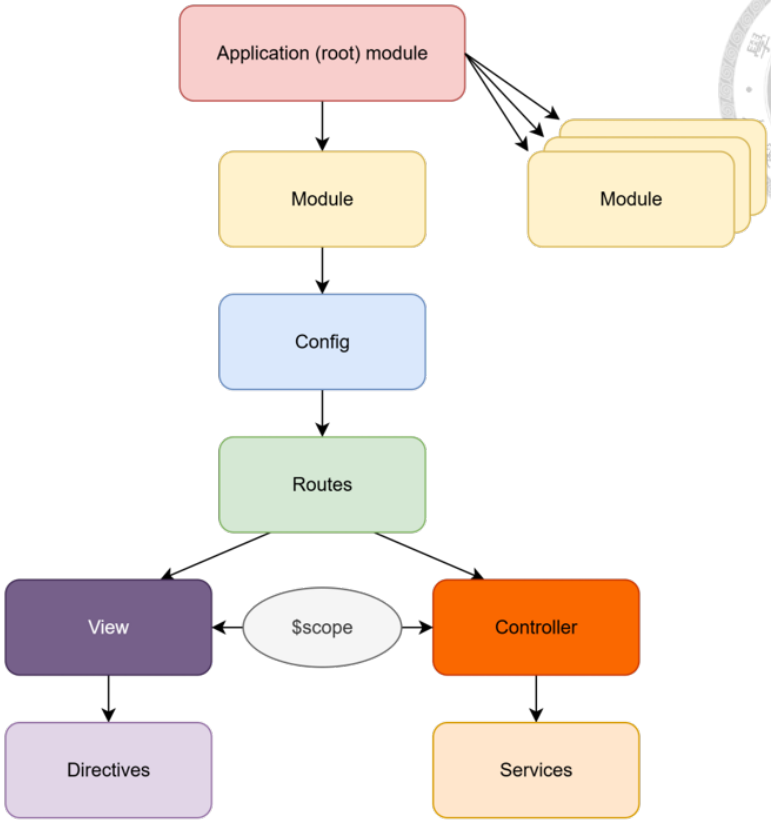


Figure 2.1: AngularJS Architecture

2.3.2 Angular

To address the limitations of AngularJS in large-scale application development, Google undertook a complete rewrite of the framework and officially released Angular 2 in 2016. From this version onward, the framework was renamed simply as "Angular" to clearly distinguish it from AngularJS. Angular adopts modern architectural design and development technologies, aiming to provide a more stable, efficient, and scalable solution to meet the needs of enterprise-level applications.

Angular uses TypeScript as its primary development language, combining a component-based architecture with a modular system, making the development of large and complex web applications more stable, flexible, and maintainable. Angular has the following key

doi:10.6342/NTU202503567

features:

Component-based Architecture

Unlike AngularJS, which used controllers and templates, Angular is centered around components. Each component encapsulates its own HTML, CSS, and logic, enhancing modularity and reusability.

TypeScript-based Development

TypeScript is a superset of JavaScript that supports static typing and object-oriented features. It improves code readability and enables better error checking during development.

Modular System

Angular offers a module system to organize applications, facilitating collaboration and maintenance, while also supporting lazy loading and optimizing performance.

Powerful CLI and Strong Official Support

Angular provides the Angular CLI for quickly generating, building, and deploying projects. It also benefits from a robust official ecosystem and long-term support.

Angular carries forward the core philosophy of AngularJS but has been completely restructured using modern technologies and architecture. With a focus on scalability, performance, and maintainability for large-scale applications, Angular has become one of the leading choices for enterprise-level frontend development.

doi:10.6342/NTU202503567

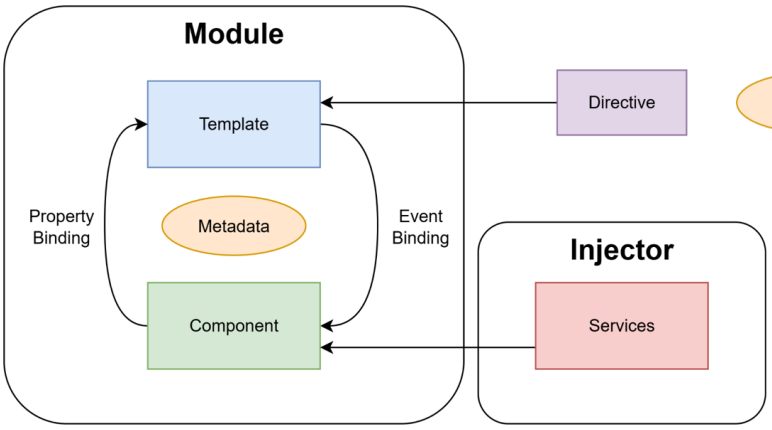


Figure 2.2: Angular Architecture

2.4 Parser

Parser is a core component in programming language processing and compiler design. Its primary function is to analyze the source code according to specific grammatical rules and convert it into a structured syntax tree or other intermediate representations for further processing.

Typically operating after the lexer, the parser receives a sequence of tokens generated by the lexer. It then checks whether the overall statements conform to the syntax rules of the programming language. If the syntax is valid, the parser assembles these tokens into an Abstract Syntax Tree, which represents the structure and logic of the program. This tree allows the computer to better understand and process the code in subsequent phases.

In our approach, we utilize three different parsers to analyze frontend projects and store the resulting ASTs in JSON format. These three parsers are: parse5 [8], @babel/parser [9], and @typescript-eslint/parser [10].

doi:10.6342/NTU202503567

2.4.1 parse5

parse5 is a JavaScript library for parsing HTML, providing a standards-compliant HTML parser. It can convert HTML strings into a DOM-like tree structure and supports multiple output formats, including HTML, XML, and JSON. Designed to be high-performance, stable, and easy to use, parse5 serves as a powerful tool for developers to process and analyze various types of web content efficiently.

```
<button ng-click="close()">
  <i aria-hidden="true"></i>
</button>
```

▲ Source Html code

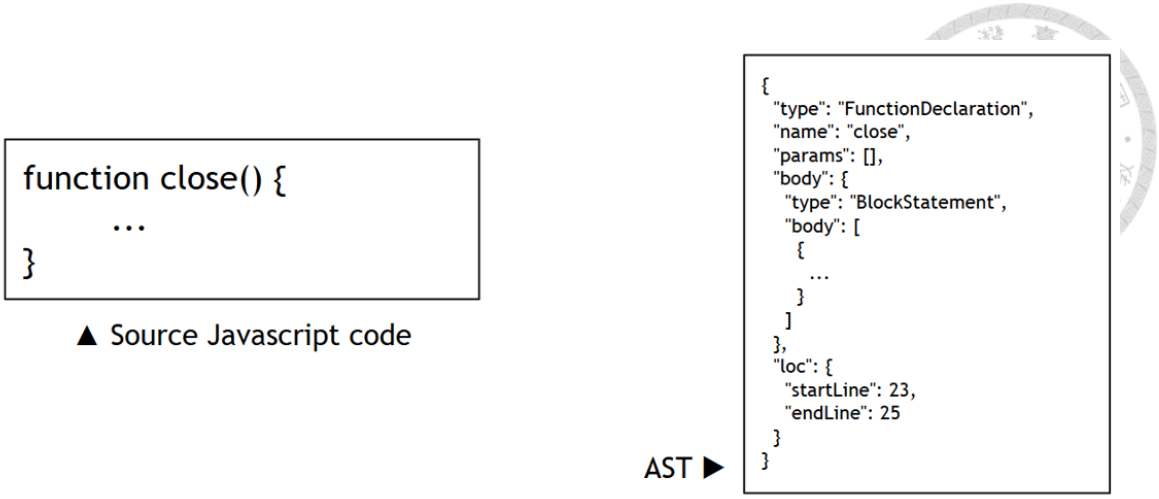
AST ▶

```
{
  "nodeName": "button",
  "attrs": [
    { "name": "ng-click", "value": "close()" },
      ...
  ],
  "sourceCodeLocation": {
    "startLine": 2,
    "endLine": 4,
    ...
  },
  "childNodes": [
    {
      "nodeName": "i",
      "attrs": [ ... ],
      "sourceCodeLocation": {...},
      "childNodes": []
    }
    ...
  ]
}
```

2.4.2 @babel/parser

@babel/parser (formerly known as Babylon) is a JavaScript parser used within the Babel toolchain. It provides a parsing mechanism that converts raw source code into an Abstract Syntax Tree. Babel [11] itself is a comprehensive toolchain primarily used to transform ECMAScript 2015 and later versions of code into backward-compatible JavaScript, enabling it to run smoothly on both modern and older browsers or runtime environments.

doi:10.6342/NTU202503567



2.4.3 @typescript-eslint/parser

@typescript-eslint/parser is a parser specifically designed for ESLint [12]. By integrating with the TypeScript ESTree, it enables ESLint to lint TypeScript code. This parser transforms TypeScript source code into an Abstract Syntax Tree that conforms to the ESTree specification, allowing developers to effectively apply ESLint's static analysis and code style rules within TypeScript projects. As a result, it helps improve code consistency and maintainability.

doi:10.6342/NTU202503567

```
ngOnInit() {
        ...
}
```

▲ Source Typescript code

AST ▶


```
{
  "type": "MethodDefinition",
  "key": {
    "type": "Identifier",
    "name": "ngOnInit",
    "loc": { "startLine": 17, "endLine": 17 }
  },
  "kind": "method",
  "value": {
    "type": "FunctionExpression",
    "async": false,
    "params": [],
    "body": {
      "type": "BlockStatement",
      "body": [
        {
          "type": "...",
          "...": "..."
        }
      ]
    }
  },
  "loc": { "startLine": 17, "endLine": 19 }
}
```


doi:10.6342/NTU202503567

Chapter 3 Overall System Architecture

The proposed system aims to implement an automated use case model generating process for frontend projects based on Angular and AngularJS. The overall architecture consists of several key stages:

First, the system takes frontend source code as input—including TypeScript, JavaScript, and HTML files from both Angular and AngularJS projects. The analysis of relationships among these frontend languages serves as the foundation for constructing the use case model.

To comprehensively parse various programming languages, the system employs multiple parsers: parse5 for parsing HTML code, @babel/parser for parsing JavaScript code, and @typescript-eslint/parser for parsing TypeScript code. These parsers convert the source code into Abstract Syntax Trees, offering a structured representation of the source code.

Next, the system traverses the AST nodes using the Visitor Pattern, extracting key information such as classes, methods, call relationships, event bindings, and component dependencies. The extracted information is then organized and stored in a relational

doi:10.6342/NTU202503567

database, forming a detailed and queryable dependency database.

Based on the database, the system further analyzes the dependencies among components and methods to automatically generate a use case model. This model includes multiple Use Case Specifications and a Use Case Diagram, clearly illustrating user interactions, system behavior, and the internal structure of the application. The resulting use case model is intended for subsequent task model prediction using the T5 model [13], though that aspect is beyond the scope of this thesis.

Overall, the system integrates multi-language parsing, structured data storage, and semantic analysis to provide a fully automated solution that bridges source code and use case modeling. This approach significantly enhances requirement comprehension, documentation generation, and maintenance efficiency in large-scale frontend projects.

doi:10.6342/NTU202503567

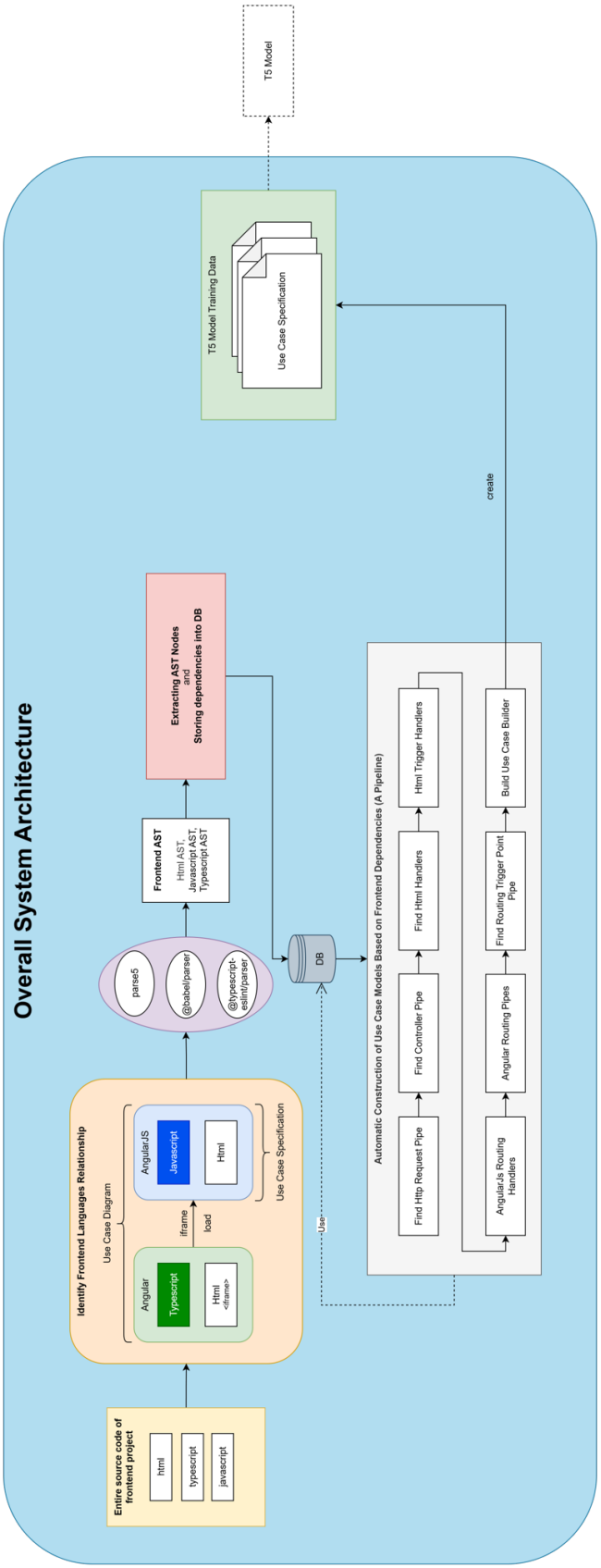


Figure 3.1: Overall System Architecture

doi:10.6342/NTU202503567

Chapter 4 Frontend Languages Relationship

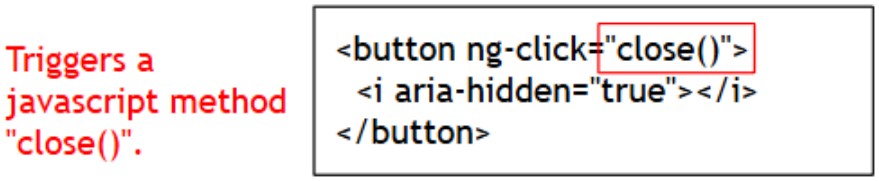
In projects based on Angular and AngularJS, the frontend source code typically consists of three programming languages: TypeScript, JavaScript, and HTML. To perform effective reverse engineering analysis on such source code, it is essential to first understand the interrelationships and role distribution among these three languages. This section explores how they interact within the frontend architecture and, based on this understanding, analyzes how to identify elements from the source code that correspond to the use case model. Furthermore, it explains how to distinguish which code fragments can serve as the basis for use case specifications, and which ones contribute to constructing the use case diagram.

4.1 Relationship between HTML and Javascript

HTML is typically responsible for triggering methods defined in JavaScript or displaying values assigned by JavaScript. In other words, HTML serves as the main component of the user interface, handling user interactions such as button clicks or input field entries and passing these events to the underlying logic functions in JavaScript. The data or state processed by JavaScript is then dynamically reflected on the screen, enabling

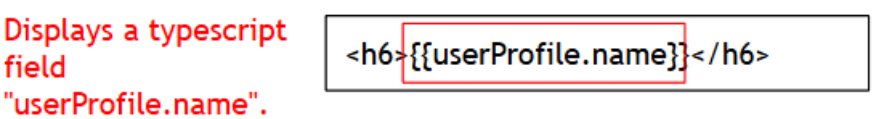
doi:10.6342/NTU202503567

real-time interaction and updates between the data and the interface. In the AngularJS framework, the interaction between HTML and JavaScript is further enhanced through special directives such as ngClick for event binding and ngModel for two-way data binding. This allows the data model in the controller to stay synchronized with the UI elements and improves the dynamic and interactive user experience.



4.2 Relationship between HTML and Typescript

Similar to the interaction between HTML and JavaScript, there is also a close relationship between HTML and TypeScript. When using the Angular framework, TypeScript is responsible for defining the logic and data structures within components, while the HTML template is used to present this data and trigger corresponding events. User actions in the HTML, such as clicking a button or entering data, are passed to functions in TypeScript through event binding. Conversely, variables or properties defined in TypeScript can be dynamically reflected in the attributes of HTML elements through property binding, enabling real-time synchronization between data and the user interface.



doi:10.6342/NTU202503567

4.3 Relationship between Javascript and Typescript

In modern frontend software architecture, TypeScript, as a statically typed superset of JavaScript, is widely used in the development of Angular version 2 and beyond. It provides stricter syntax checking and modular design, which enhances maintainability and scalability in large-scale projects. In contrast, JavaScript remains the primary language for AngularJS (that is, Angular 1.x), whose architecture and design philosophy differ significantly from later versions of Angular.

To enable a smooth transition and coexistence between legacy and modern systems, various technical approaches can be employed for their integration. A common strategy is to embed a standalone AngularJS application within an Angular application using an `<iframe>` element. In this setup, the TypeScript-based Angular application handles overall routing for the system, while the embedded AngularJS application uses UI-Router within the iframe to manage hash-based URL routing and dynamically inject the corresponding views and controllers based on the application state.

This architecture not only effectively isolates the execution environments of the two technology stacks, reducing coupling between them, but also ensures system compatibility and stability.

In summary, although TypeScript and JavaScript belong to different generations of frontend technologies, they can coexist harmoniously within the same system through integration methods such as iframe embedding and routing coordination. This supports strategic flexibility and continuous evolution for enterprises facing technology upgrades.

doi:10.6342/NTU202503567

4.4 Modeling Use Case Model

AngularJS is responsible for handling the core logic of the entire project. Therefore, we analyze the interaction between JavaScript and HTML to extract and derive the system's Use Case Specifications. This approach effectively reveals the interaction details between the user and the system and captures the key functional flows.

In addition, we take into account the relationship and collaboration between Angular and AngularJS, leveraging their architectural division of responsibilities and integration mechanisms to construct the overall Use Case Diagram. This multi-layered analysis enhances the understanding of system behavior, thereby providing a solid theoretical foundation for subsequent system design.

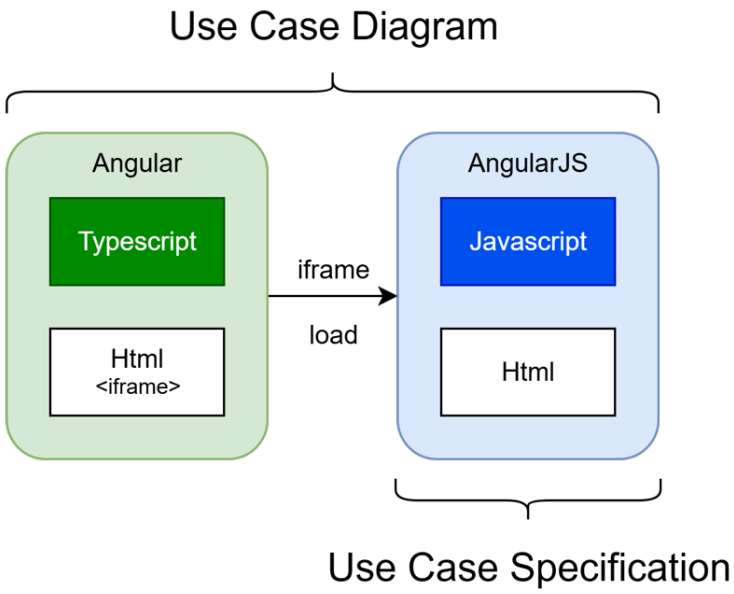


Figure 4.1: Modeling Use Case Model Based on Frontend Languages Relationship

doi:10.6342/NTU202503567

Chapter 5 Extraction of AST Nodes

This chapter explains how to extract abstract syntax tree nodes from TypeScript, JavaScript, and HTML source code, and how to design appropriate data models based on the syntactic features of each language to support subsequent storage and querying.

To effectively convert the structure of abstract syntax tree nodes into a form that can be represented in a database, this chapter presents separate entity relationship diagrams for the three frontend programming languages. In addition, a class diagram specifically designed for the abstract syntax tree extraction process is provided.

5.1 Entity-Relationship Diagram Design for TypeScript

Figure 5.1 presents the entity relationship diagram designed for the syntax and structure of TypeScript. Each entity and relationship in the diagram corresponds to common language elements found in the abstract syntax tree. The purpose of this design is to comprehensively represent the structural relationships among syntax units in TypeScript, such as classes, methods, properties, and decorators, and to serve as the foundation for extracting nodes and storing them in the database. Detailed explanations of certain key entities will be provided below, focusing on structures that are highly relevant to system functionality. Simpler entities with straightforward structures will not be further elaborated.

doi:10.6342/NTU202503567

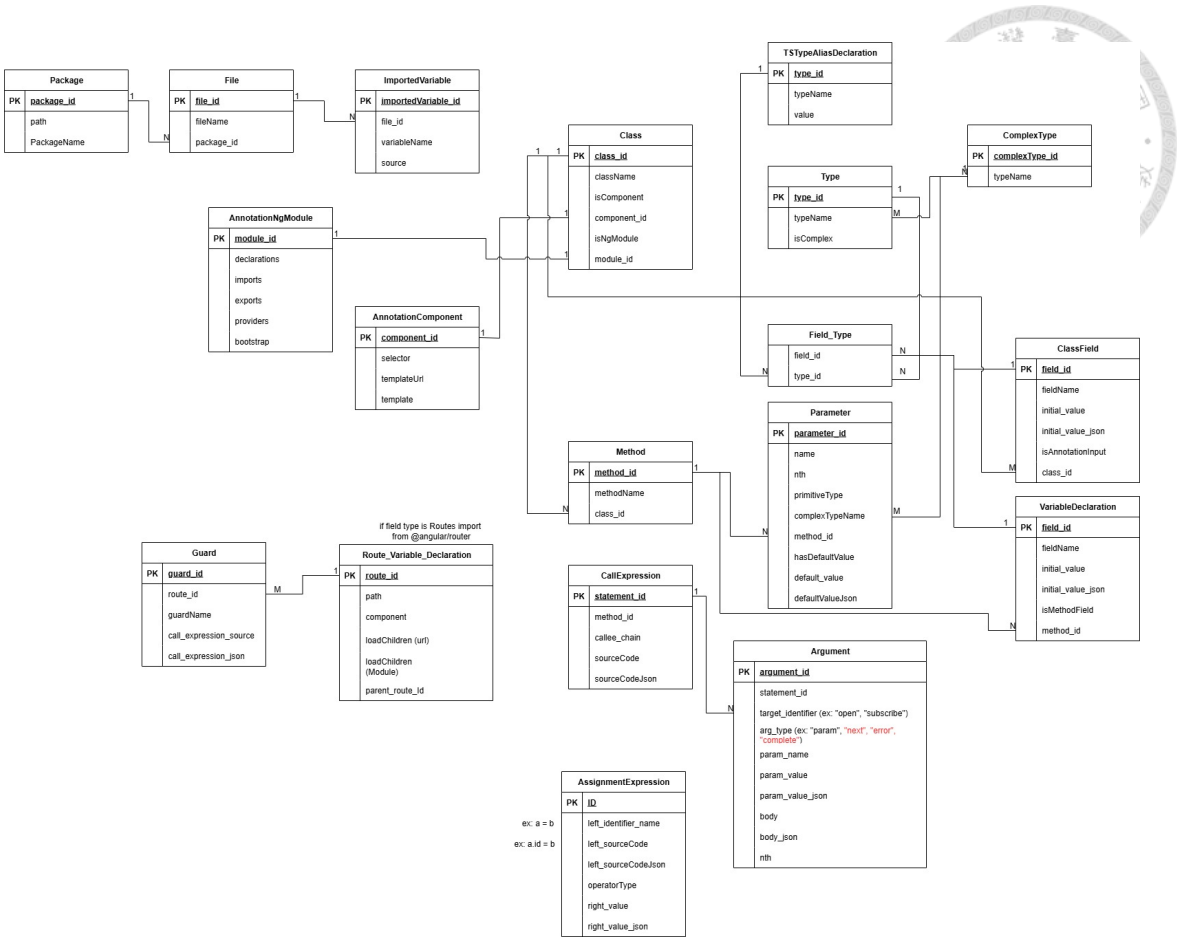


Figure 5.1: ER Diagram Design for Typescript

Angular Routing

Angular routing is responsible for configuring the routing rules of an application, defining the mapping between URLs and components. It supports nested routes, route guards, and lazy loading of modules. Common fields include **path**, **component**, **loadChildren**, **canActivate**, and **children**. The table below explains the purpose and design intention of each field:

doi:10.6342/NTU202503567

Field	Description	Purpose & Usage
path	Defines the URL path corresponding to the route (relative to its parent route). For example, `path: 'dashboard'` corresponds to `/dashboard`.	Used to specify which route configuration should be matched when a user navigates to a particular URL. `''` represents the default path, commonly used for the homepage or parent route.
component	Specifies the component to be displayed when this route is activated. The component will be rendered inside the `<router-outlet>`.	Used to control the UI that should be presented on the screen.
loadChildren	Specifies the method for lazy loading a module, typically using the ES6 `import()` syntax.	To improve application performance, the corresponding module is loaded only when the route is actually navigated to, reducing the initial download size.
canActivate	Specifies a set of route guards used to determine whether the user has permission to access the route. It can be a single guard or multiple guards.	Provides security control, such as login authentication, permission checks, or redirecting to another page (for example, guiding the user to a settings page if setup is incomplete).
children	Declares an array of nested child routes, allowing sub-routes to be defined under the current route.	Enables a hierarchical routing structure, commonly used for wrapping subpages with a layout or dividing multiple subfeatures within a module.

Table 5.1: Description of Common Fields in Angular Routing

doi:10.6342/NTU202503567

Based on the common field settings of Angular Routing, we designed the following ER Diagram as shown below:

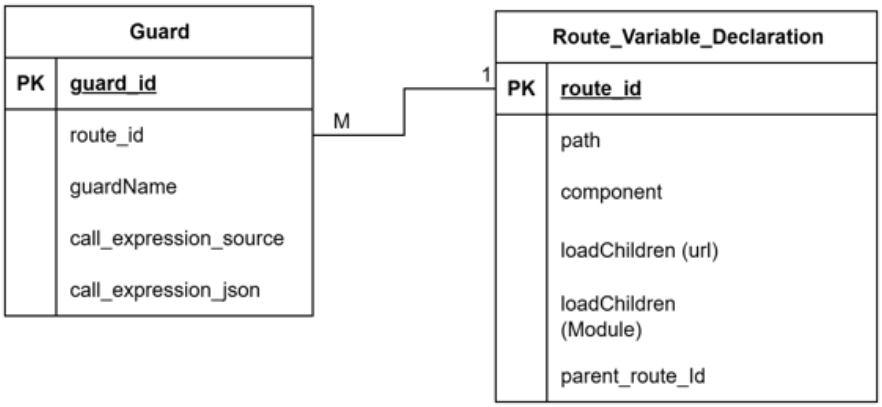


Figure 5.2: ER Diagram Design for Angular Routing

@NgModule

In Angular, `@NgModule` is a decorator used to define a module. It is responsible for organizing components, services, importing other modules, and bootstrapping the application startup. Common configuration fields include **declarations**, **bootstrap**, **imports**, **exports**, and **providers**. The table below explains the purpose and design intention of each field:

doi:10.6342/NTU202503567

Field	Description	Purpose & Usage
declarations	Declares the components, directives, and pipes that belong to this module.	Informs Angular that these classes are part of the module and can be used and compiled within it.
imports	Imports other modules, allowing this module to use the components and features they export.	Facilitates sharing of components, services, and functionalities between modules, enabling a modular architecture.
exports	Exports the components, directives, or pipes from this module for use in other modules.	Allows other modules to import and use the elements exported by this module.
providers	Used to register services required by this module, enabling Angular's dependency injection system to provide these services to components or other services within the module.	Configures the scope and lifecycle of services, controlling the behavior of dependency injection.
bootstrap	Specifies the root component to load when the application starts.	Primarily used in the root module to bootstrap the entire Angular application.

Table 5.2: Description of Common Fields in @NgModule

Based on the common field settings of `@NgModule`, we designed the following ER Diagram as shown below:

doi:10.6342/NTU202503567

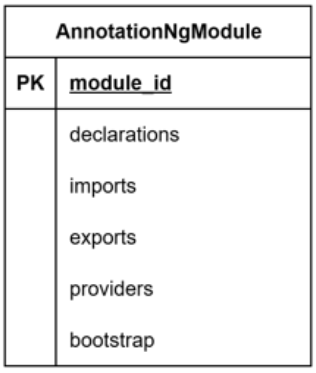


Figure 5.3: ER Diagram Design for @NgModule

@Component

In Angular, `@Component` is a decorator used to define a UI component. It is responsible for specifying the component's selector, template, styles, and other configurations. Common fields include **selector**, **templateUrl**, and **template**. The table below explains the purpose and design intention of each field:

doi:10.6342/NTU202503567

Field	Description	Purpose & Usage
selector	Defines the HTML tag name of the component, allowing it to be used in other pages.	Enables developers to embed the component into other templates using custom tags (for example, `<app-header>`).
templateUrl	Specifies the path to the template file that defines the component's view content.	Separates the template content into an external file, which helps improve code organization and readability.
template	Defines the HTML template content directly as a string.	Suitable for simple components or scenarios where templates are generated dynamically, avoiding the need to create separate files.

Table 5.3: Description of Common Fields in @Component

Based on the common field settings of `@Component`, we designed the following ER Diagram as shown below:

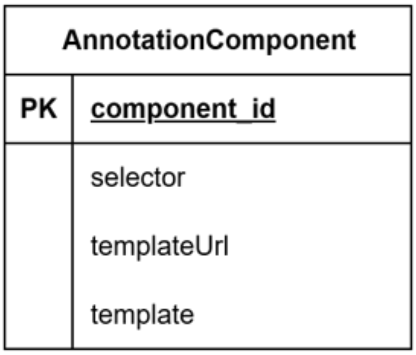


Figure 5.4: ER Diagram Design for @Component

doi:10.6342/NTU202503567

5.2 Entity-Relationship Diagram Design for JavaScript

Figure 5.5 presents the entity relationship diagram designed for the syntax and structure of JavaScript. Each entity and relationship corresponds to common language elements found in the abstract syntax tree. The diagram aims to comprehensively represent the structural relationships of basic JavaScript syntax as well as AngularJS-specific syntax units, such as controllers, modules, services, and directives. It serves as the foundation for subsequent node extraction and storage in the database. Detailed explanations of certain key entities will be provided in the following discussion, focusing on structures highly relevant to system functionality. Simpler entities with straightforward structures will not be further elaborated.

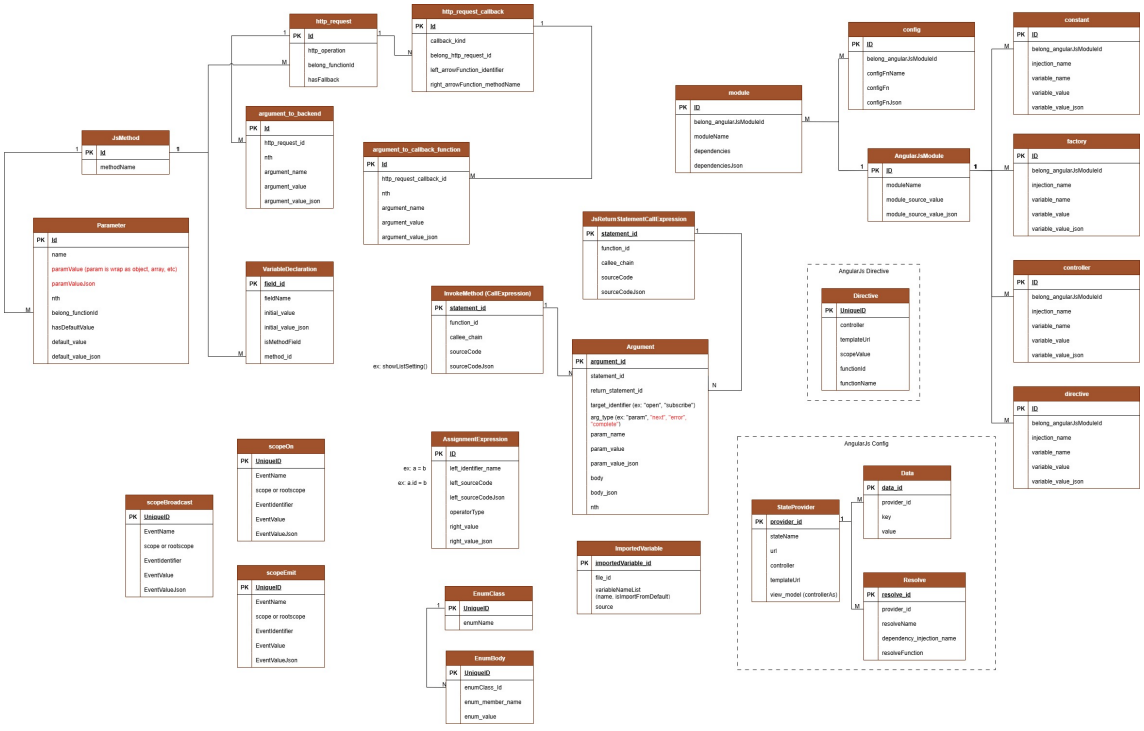


Figure 5.5: ER Diagram Design for Javascript

AngularJS Module

doi:10.6342/NTU202503567

In AngularJS, module is the core unit used to define an application module. It is responsible for organizing controllers, services, constants, route configurations, directives, and other functionalities. Modules can depend on other modules, forming a hierarchical structure that promotes modularity and code reuse. Below is an example of a typical AngularJS module definition:

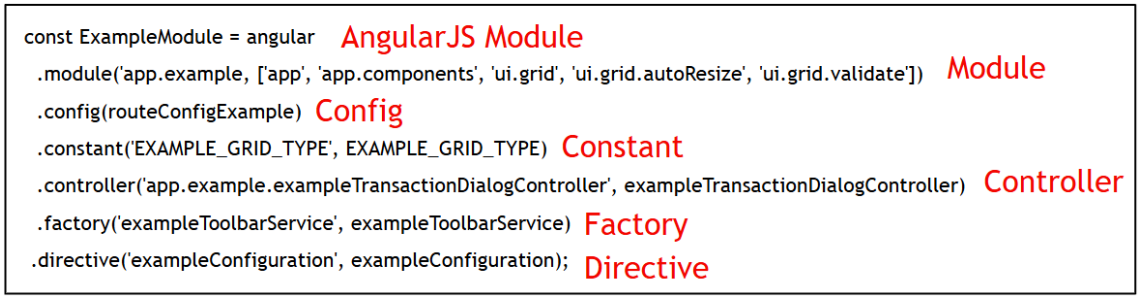


Figure 5.6: Example of AngularJS Module Configuration

The table below explains the purpose and design intention of each field:

doi:10.6342/NTU202503567

Field	Description	Purpose & Usage
module	Creates or extends an AngularJS module and sets its dependent modules.	Used to organize application functionality into modular units and support dependency management between modules.
config	Configures the module's configuration phase, typically used for route setup.	Initializes module behavior, such as setting routing rules.
constant	Declares immutable constant values for injection within the module.	Provides global constants to facilitate shared configuration values across multiple places.
controller	Defines a controller responsible for managing the data and behavior of the view.	Controls the UI logic and user interactions.
factory	Creates a factory service responsible for encapsulating reusable business logic.	Implements an injectable service that provides data processing and business logic.
directive	Defines a directive used to extend HTML functionality and manipulate the DOM.	Creates custom elements or attributes to enhance UI flexibility and reusability.

Table 5.4: Description of Common Fields in AngularJS Module

Based on the common field settings of modules, we designed the following ER Diagram as shown below:

doi:10.6342/NTU202503567

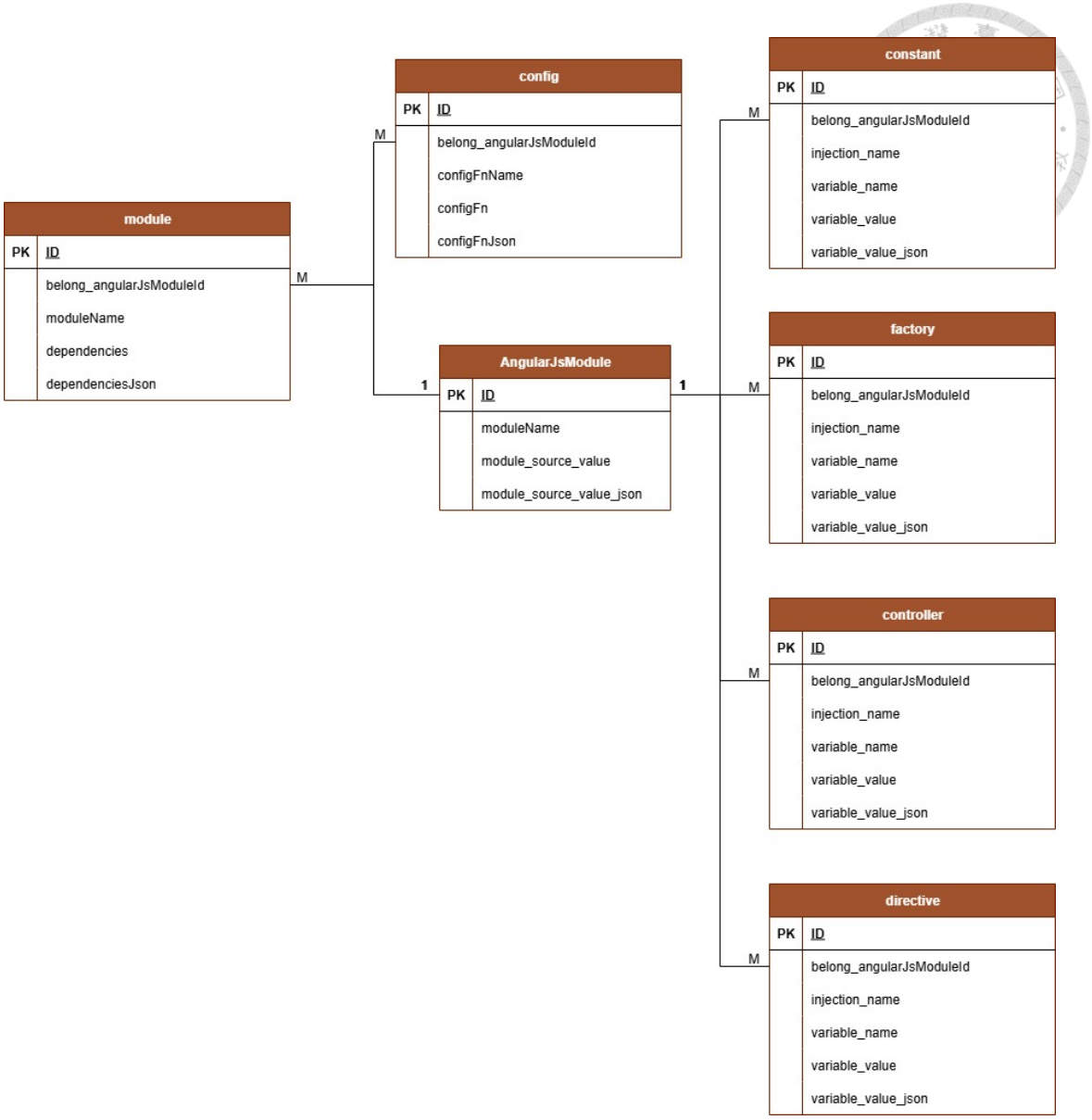


Figure 5.7: ER Diagram Design for AngularJS Module

AngularJS Directive

In AngularJS, directive is a core feature used to extend HTML syntax, allowing developers to customize DOM behavior, create reusable UI components, or control the presentation of UI logic. Directives encapsulate templates and logic and can be embedded into pages as attributes or elements. Below is an example of a typical custom directive definition in AngularJS:

doi:10.6342/NTU202503567

```
function XXXDirective() {
  'ngInject';

  return {
    scope: {
      editMode: '=',
      entityId: '@',
      refresh: '&',
    },
    restrict: 'E',
    controller: 'app.components.XXXController',
    templateUrl: 'scripts/components/relationships/XXX.directive.html'
  };
}
```

Figure 5.8: Example of AngularJS Directive Definition

The table below explains the purpose and design intention of common fields:

Field	Description	Purpose & Usage
controller	Specifies the controller of the directive, responsible for handling template logic and external interactions.	Encapsulates operational logic within the controller to enable data binding and event handling.
templateUrl	Specifies the path to the directive's template HTML file.	Separates the template from the logic, enhancing maintainability and readability.
scope	Defines the directive's scope and its data binding methods with external components.	Controls data communication between the directive and external components, such as two-way binding (=), text binding (@), or method binding (&).

Table 5.5: Description of Common Fields in AngularJS Directive

doi:10.6342/NTU202503567

Based on the common field settings of directives, we designed the following ER Diagram as shown below:

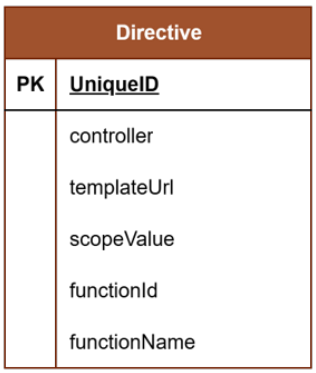


Figure 5.9: ER Diagram Design for AngularJS Directive

AngularJS Config

In AngularJS, the `config` phase is commonly used to configure routing behavior. Through UI-Router's `$stateProvider.state` method, various application states can be declared. Each state definition includes the state's name, corresponding URL, controller, controller alias, static data, and functions within `resolve`. These `resolve` functions are used to pre-fetch necessary data or initialize methods to ensure that related resources are ready before entering the state, with the results injected into the controller for use. Below is a typical example:

doi:10.6342/NTU202503567

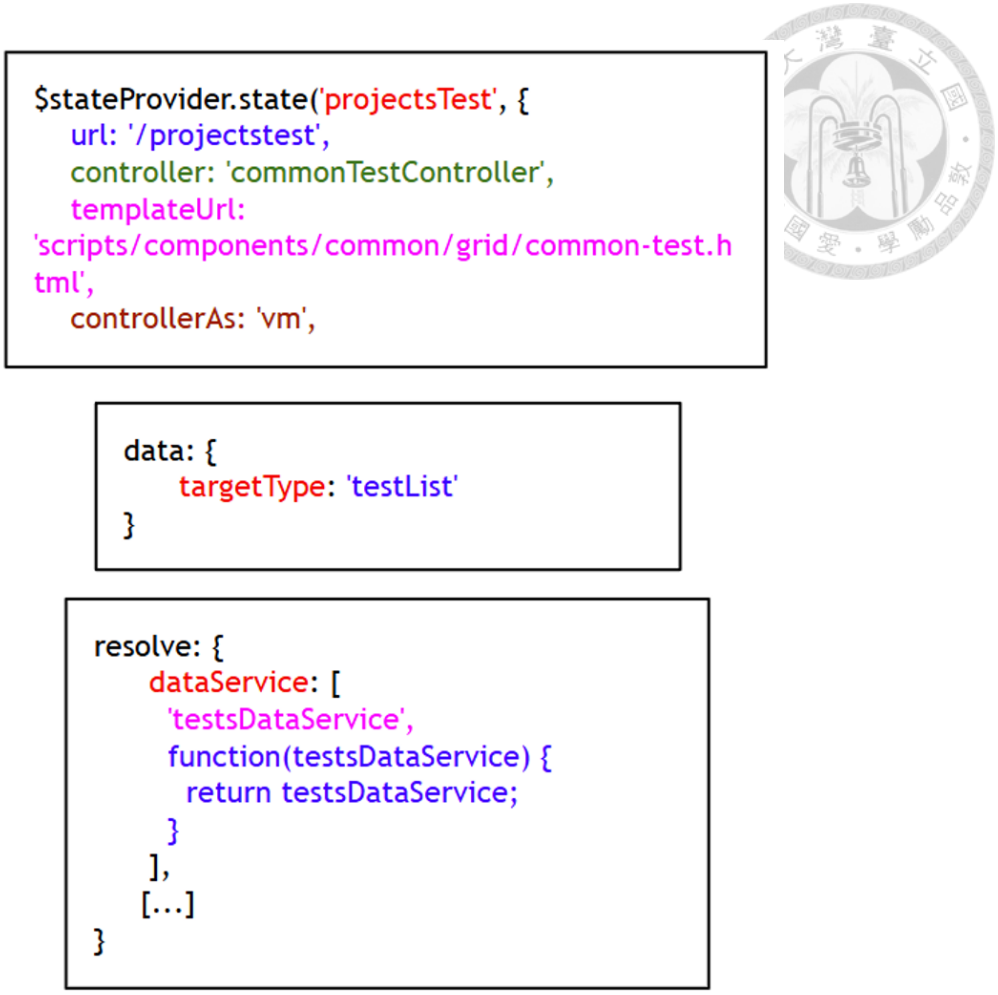


Figure 5.10: Example of AngularJS Config

The table below explains the purpose and design intention of each configuration field:

doi:10.6342/NTU202503567

Field	Description	Purpose & Usage
stateName	Defines the state name used for identification and navigation within the routing system, supporting view transitions inside the application.	Serves to mark a logical unit of a state.
url	Defines the URL path corresponding to the state.	Allows users to navigate to the specified state via the URL.
controller	Specifies the controller associated with the state.	Responsible for handling the page logic and data binding.
templateUrl	Specifies the path to the template HTML used by the state.	Responsible for displaying the UI of the view.
viewModel (`controllerAs`)	Specifies the alias used by the controller within the template.	Enhances readability and consistency, avoiding direct manipulation of `$scope`.
Key-value pairs in `data`	Declare static data available for the state.	Typically used to set permissions, page titles, category types, and other static information.
resolveName in `resolve`	Specifies the name of the data item to be resolved before state activation.	This name serves as the variable identifier injected into the controller for accessing the resolved data.
Dependency injection name in `resolve`	Lists the names of services to be injected as parameters into the resolve function.	Ensures that the necessary services are properly injected and available for data fetching or processing before entering the state.
resolveFunction in `resolve`	Defines the function that fetches or processes data, usually by calling injected services.	Executes the logic to prepare required data, guaranteeing it is ready before the state transition completes.

Table 5.6: Description of Common Fields in AngularJS Config

doi:10.6342/NTU202503567

Based on the above state configuration fields, we designed the following ER Diagram as shown below:

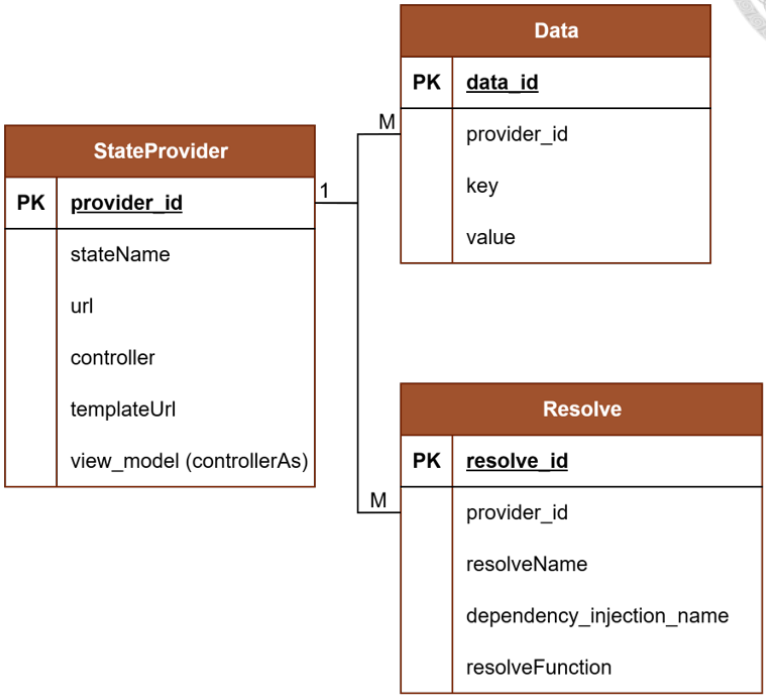


Figure 5.11: ER Diagram Design for AngularJS Config

5.3 Entity-Relationship Diagram Design for HTML

HTML elements can be mainly divided into two categories: one is native HTML tags, such as input fields `<input>`, buttons `<button>`, headings `<h2>`, and so on; the other consists of custom tags defined through Angular's selector or AngularJS's directive. These custom tags encapsulate complex logic and UI into reusable components.

In Angular, developers can define components with a `selector` using the `@Component` decorator, for example:

```
@Component({
  selector: 'app-user-setting',
  templateUrl: './user-setting.component.html'
```


doi:10.6342/NTU202503567

```
})
export class UserSettingComponent {}
```

When <app-user-setting></app-user-setting> appears in a template, Angular parses it as the corresponding component's content.

In AngularJS, custom tags are implemented through directive, for example:

```
app.directive('customButton', function() {
  return {
    restrict: 'E',
    template: '<button>Click Me</button>'
  };
});
```

At this point, <custom-button></custom-button> can be used directly in the HTML, and AngularJS will automatically translate it into the corresponding template content.

For native HTML tags, we designed the following fields:

Id：A unique identifier corresponding to each element.

loc：The position of the element in the source code (line and column information).

fileId：The identifier of the file to which the element belongs.

attributes：All attributes of the element, recorded as key-value pairs. For example, will be recorded as: key: ng-if, value: true.

doi:10.6342/NTU202503567

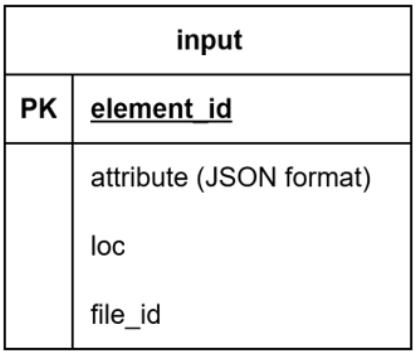


Figure 5.12: ER Diagram Design for Native HTML Element

For custom elements, the data structure we designed retains all fields of native elements and additionally includes a record of the custom tag name:

Id：A unique identifier for each element.

loc：The location information where it appears in the source code.

fileId：The identifier of the file to which it belongs.

attributes：All attribute information stored in key-value pair format.

tag-name：The name of the custom tag (for example, app-user-setting, custom-button, etc.).

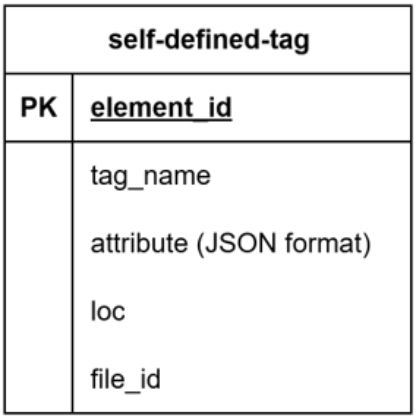


Figure 5.13: ER Diagram Design for Custom HTML Element

doi:10.6342/NTU202503567

This classification and design can assist in accurately matching and correlating components used within Angular systems in subsequent analysis.

5.4 Class Diagram Design for AST Node Extraction

The AST node extraction process consists of the following three main steps:

Step 1. The Non-static Factory Pattern is employed to dynamically generate corresponding node processors for HTML, JavaScript, and TypeScript, addressing the traversal requirements of abstract syntax tree nodes with different language characteristics.

Step 2. The processors recursively traverse AST nodes and use the Non-static Factory Pattern to generate corresponding node instances, ensuring that each node type has a corresponding object representation.

Step 3. The Visitor Pattern is used to visit these node instances, extract the information that needs to be stored in the database, and create the corresponding node entities.

doi:10.6342/NTU202503567

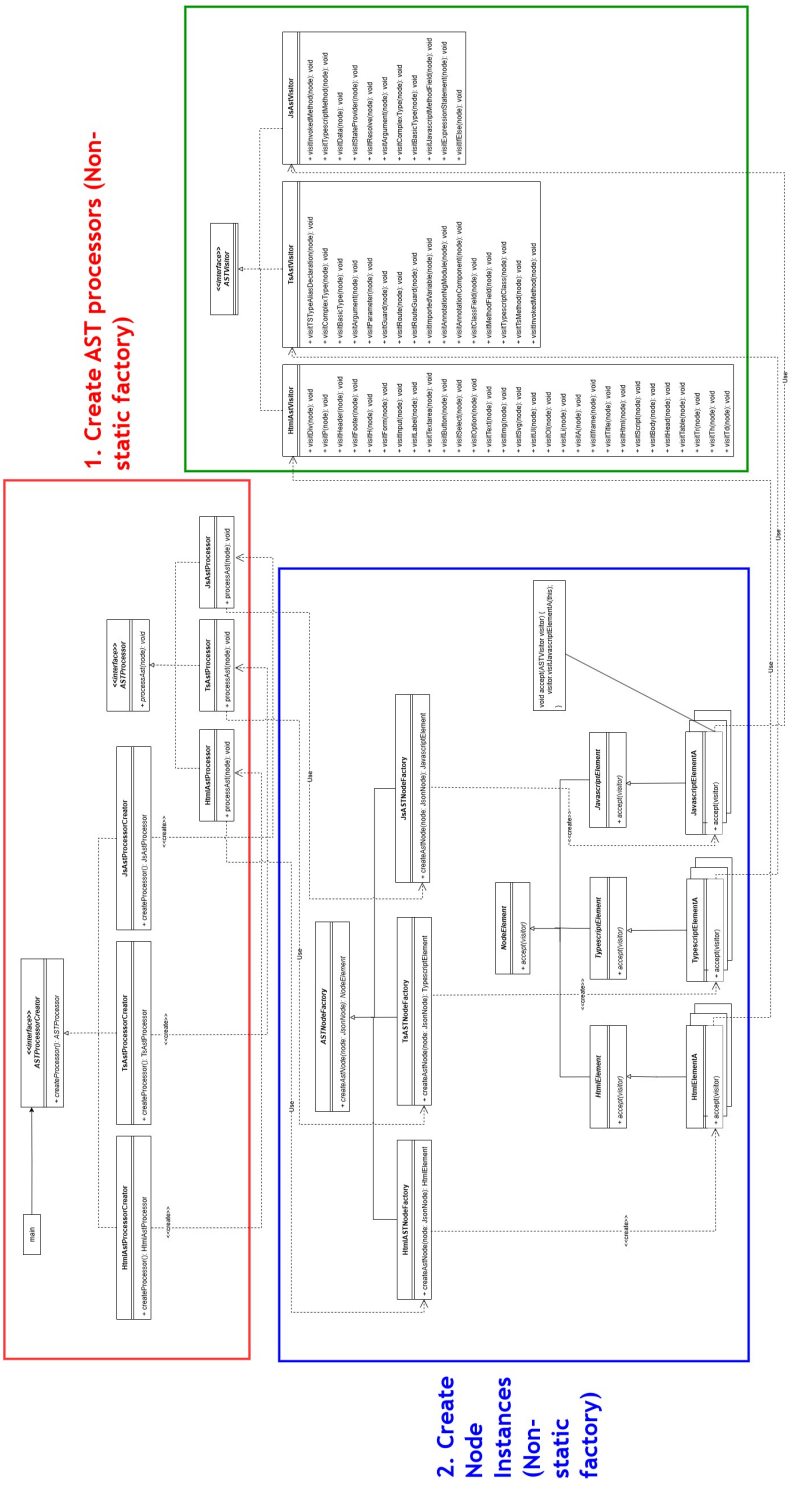


Figure 5.14: Class Diagram Design for AST Node Extraction

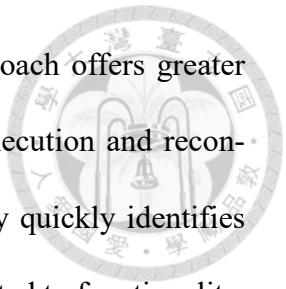
doi:10.6342/NTU202503567

Chapter 6 Automatic Construction of Use Case Models Based on Frontend Dependencies

When attempting to understand how users interact with a web application, merely reading the source code is often insufficient to grasp the underlying logic. To address this, this chapter proposes an algorithm that statically analyzes dependencies within the frontend code to automatically construct a use case model, thereby reconstructing the actual usage scenarios and workflows of the system.

The algorithm adopts a bottom-up analysis approach, starting from HTTP requests in the frontend code and tracing upward layer by layer to the root of the project. The choice to begin with HTTP requests is based on the fact that, in typical REST architectures, nearly every HTTP request corresponds to a concrete functional requirement such as ”creating an order,” ”querying data,” or ”modifying settings.” When a user triggers an action on the interface that results in a request being sent, their intent is clearly expressed and handed over to the backend for processing. Therefore, tracing the code from these requests ensures that the analyzed execution paths carry clear semantics and correspond directly to actual system functions.

doi:10.6342/NTU202503567



Compared to a top-down analysis strategy, the bottom-up approach offers greater focus and efficiency. By starting from actual call expressions in execution and reconstructing the flow through function call chains, this method not only quickly identifies core logic but also effectively filters out structural noise that is unrelated to functionality, such as common initialization settings in large frameworks. Through this strategy, the system can more accurately and efficiently extract hidden usage scenarios within frontend code and use them to construct concrete use case models.

6.1 Algorithm

The use case model construction algorithm proposed in this study adopts the **Pipeline pattern**, combining **depth-first search** and **breadth-first search**. Starting from HTTP requests in the frontend code, the algorithm traces upward layer by layer to the root of the project in order to uncover all potential usage scenarios hidden within the system. The overall process is divided into eight main steps, sequentially tracking the dependency paths within the frontend system. Based on the obtained dependency paths, a complete use case model is constructed. The steps are as follows:

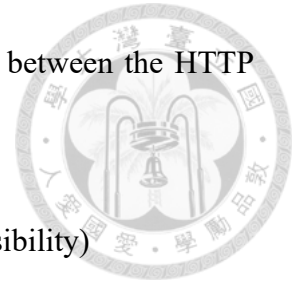
Step 1. Identify All HTTP Requests

Scan all HTTP requests present in the entire frontend project. If a request has a successful callback (such as `.then()` or `.subscribe()`), recursively trace the callback function to check whether it contains other HTTP requests, in order to capture chained request behaviors.

Step 2. Find All Functions in the Controller That Call the HTTP Request

Use call expression analysis to identify all functions in the Controller that invoke

doi:10.6342/NTU202503567



the HTTP request, and establish the dependency relationship between the HTTP request and these functions.

Step 3. Locate the View Bound to the Controller (Chain-of-Responsibility)

Identify the view associated with the Controller to facilitate the next step, which involves locating the source of the Controller method's trigger.

Step 4. Trace the Trigger Source That Triggers the Function (Chain-of-Responsibility)

Trace the origins of the Controller function triggers, which may come from event bindings in templates (such as `ng-click`), or be called during the Controller's initialization phase (such as `$onInit`).

Step 5. Trace from the Template Back to the State Definition (Chain-of-Responsibility)

Continue tracing upward from the template to locate the template file specified in the `$stateProvider.state` definition, which serves as the entry point for the AngularJS state.

Step 6. Trace from the State Definition Back to the Application Entry Point

From three perspectives —URL resolution, `<iframe>` elements (used for embedding independent subsystems or modules), and Angular routing configurations — trace and locate the overall application entry point to construct the page navigation flow.

Step 7. Identify Route Triggers

Identify HTML elements in the template that trigger routing via [routerLink] or similar bindings. When users interact with these elements, navigation is initiated.

Step 8. Construct the Use Case Model (Builder Pattern)

Based on the dependency paths traced in the previous steps, integrate the gathered

doi:10.6342/NTU202503567

dependencies to construct the use case model. This step applies the Builder Pattern to support the composability and extensibility of the use case model.

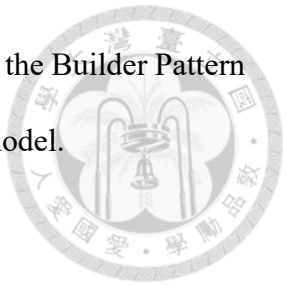


Figure 6.1 is the class diagram designed to implement the algorithm.

doi:10.6342/NTU202503567

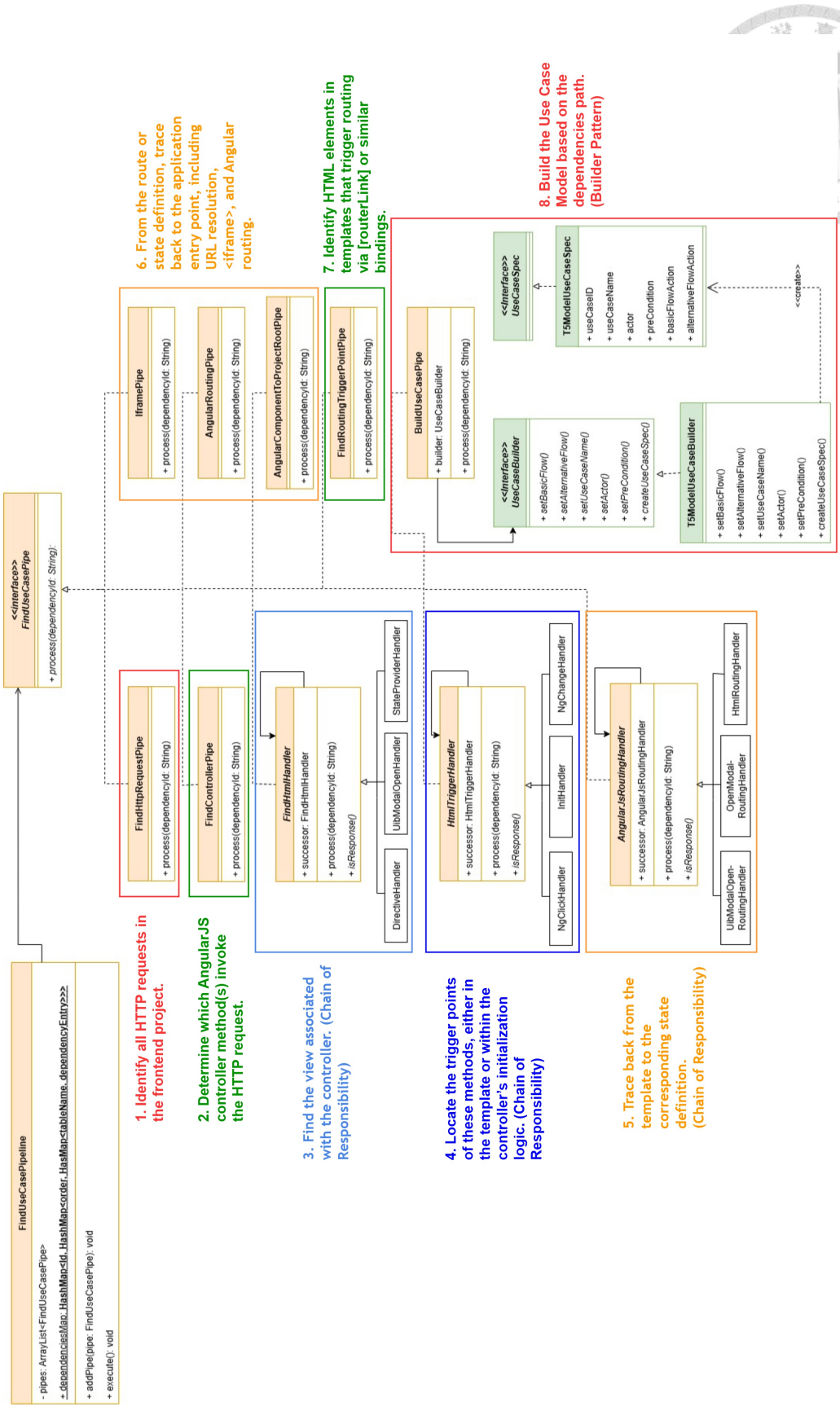


Figure 6.1: Class Diagram Design for Automatic Construction of Use Case Models Based on Frontend Dependencies

doi:10.6342/NTU202503567

6.2 Find HTTP Request Pipe

The first step of the analysis is to identify all HTTP requests within the frontend project. If a request includes a success callback function, it is necessary to further trace that callback to determine whether additional HTTP requests are invoked within it.

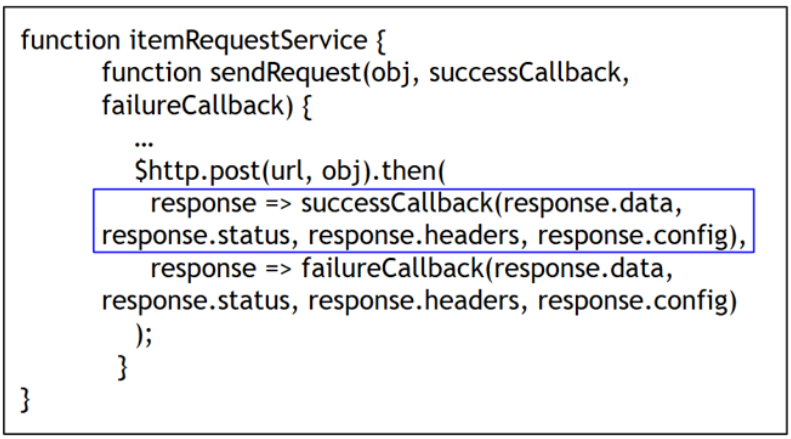
For example, in the following code snippet, the `POST` request has a success callback function. Therefore, it is essential to trace the location where `itemRequestService.validateStatus` is called, in order to determine whether its success callback further triggers other HTTP requests.

```
function itemRequestService {
        function validateStatus(obj, successCallback, failureCallback) {
            ...
            $http.post(url, obj).then(
              response => successCallback(response.data, response.status,
response.headers, response.config),
              response => failureCallback(response.data, response.status, response.headers,
response.config)
            );
          }
}
```

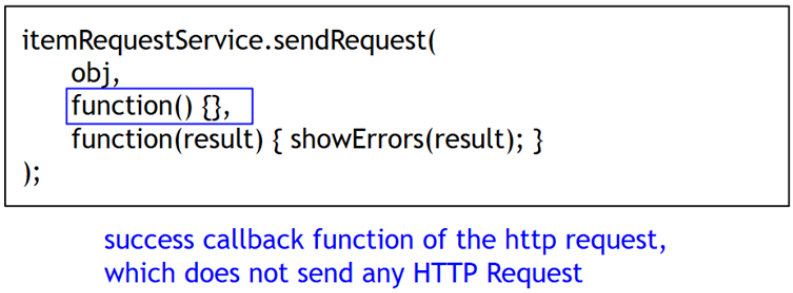
```
itemRequestService.validateStatus(  success callback function
  obj,                              of the http request
  function(data) {
    itemRequestService.sendRequest(
      obj,
      function() {},
      function(result) { showErrors(result); }
    );
  },
  function(data) { showErrors(result); }
);
```

As shown in the figure above, the success callback function invokes `itemRequestService.sendRequest`. At this point, we need to continue tracing the definition of that function to determine whether it contains any additional HTTP requests.

doi:10.6342/NTU202503567



The function `itemRequestService.sendRequest` also contains an HTTP request with a success callback function. Therefore, we must recursively continue to locate where `sendRequest` is called and analyze whether its callback function triggers additional HTTP requests, continuing this process until the end of the entire request chain is reached.



Finally, we establish the complete HTTP request dependency relationships, with the dependencies path as shown below:

```
Dependency Path Map:
Path1: (Type: HTTP Request, dependency: itemRequestService.
  ↪ sendRequest)
      → (Type: HTTP Request, dependency: itemRequestService.
        ↪ validateStatus),
Path2: ...
```


doi:10.6342/NTU202503567

6.3 Find Controller Pipe

After establishing the complete HTTP request tracing chain, the second step is to identify the Controller functions that trigger these HTTP requests based on the control flow in the MVC architecture. In AngularJS, Controllers are responsible for handling user actions and calling corresponding Service layer functions, which may include HTTP request calls.

Using the call expression analysis results constructed in the previous step, we can trace backward all the call origins of functions that contain HTTP requests. When these call origins come from Controller functions, we can establish dependency relationships between the HTTP requests and the Controller functions.

```
function ExampleIssueRequestController(..., itemRequestService, ...){
        function issueExampleRequest(){
                ...
                itemRequestService.validateStatus(...)
        }
}
```

```
function ExampleApplyResultsController(..., itemRequestService, ...){
        function issueDecomExampleRequests(){
                ...
                itemRequestService.validateStatus(...)
        }
}
```

```
Dependency Path Map:
Path1-1: (Type: HTTP Request, dependency: itemRequestService.
  ↪ sendRequest)
        → (Type: HTTP Request, dependency: itemRequestService.
          ↪ validateStatus)
        → (Type: controller function, dependency:
          ↪ issueExampleRequest
          <controller: ExampleIssueRequestController>)
```


doi:10.6342/NTU202503567

```
Path1-2: (Type: HTTP Request, dependency: itemRequestService.
  ↪ sendRequest)
        → (Type: HTTP Request, dependency: itemRequestService.
          ↪ validateStatus)
        → (Type: controller function, dependency:
          ↪ issueDecomExampleRequests
          <controller: ExampleApplyResultsController>)
 ...
```

6.4 Find HTML Handlers

This step aims to identify the view associated with a controller, thereby establishing a clear correspondence between the Controller and the View. Through this association, we can understand the actual UI scope that the controller operates on, which provides crucial basis for subsequent analysis of trigger sources (such as user clicks, data bindings, etc.). This helps us fully reconstruct the system's control flow.

Below, three common methods of view binding are introduced to explain how controllers and views are linked:

1. **UibModalOpenHandler**：Using the `$uibModal.open` method to dynamically bind views and controllers at runtime. The `templateUrl` specifies the HTML template to be loaded, while the `controller` specifies the controller responsible for the view's logic. This approach is typically used in modal dialog scenarios:

   ```
   function viewPortDetail() {
      $uibModal.open({
        templateUrl: 'xxx.html',
        controller: portModalController,
   ```


doi:10.6342/NTU202503567

```
      ...
    })
}
```

2. **DirectiveHandler**：Binding views and controllers through custom directives. During definition, specify `templateUrl` and `controller`. When used, simply include the directive tag in the HTML to apply the template content and logic to the view:

```
function auditTrailDirective() {
  'ngInject';

  return {
    templateUrl: './audit-trail.html',
    controller: auditTrailController,
    scope: {
      auditData: '='
    }
  };
}
```

3. **StateProviderHandler**：Binding views and controllers through AngularJS routing configuration ($stateProvider). When the user navigates to a specified URL, the system loads the corresponding templateUrl and activates the specified controller:

```
$stateProvider.state('projectsTest', {
    url: '/projectstest',
    controller: 'commonTestController',
    templateUrl: 'scripts/components/common/grid/common-test
      ↪ .html',
    controllerAs: 'vm',
}
```


doi:10.6342/NTU202503567

The following example uses UibModalOpenHandler to demonstrate the correspondence between Controller and View:

```
function exampleService(...) {
        function issueExampleRequest() {
                $uibModal.open({
                    templateUrl: 'example-issue-request.html',
                    controller: ExampleIssueRequestController,
                        ...
                });
        }
}
```

In the code above, `ExampleIssueRequestController` is bound to the template `example-issue-request.html` via the `$uibModal.open()` method. When `issueExampleRequest()` is called, a dialog opens that loads the specified HTML view and activates the corresponding controller. Therefore, we can confirm that the view associated with `ExampleIssueRequestController` is `example-issue-request.html`.

At this point, the Dependency Path Map expands as follows:

```
Dependency Path Map:
Path1-1-1: (Type: HTTP Request, dependency: itemRequestService.
  ↪ sendRequest)
        → (Type: HTTP Request, dependency: itemRequestService.
          ↪ validateStatus)
        → (Type: controller function, dependency:
          ↪ issueExampleRequest
          <controller: ExampleIssueRequestController>)
        → (Type: htmlFile, dependency: example-issue-request.
          ↪ html)
 ...
```


doi:10.6342/NTU202503567

6.5 HTML Trigger Handlers

To more comprehensively reconstruct the correspondence between user actions and system logic, this step further locates the trigger points of each controller method. These methods may be triggered by AngularJS directives in the template (such as `ng-click`) or may be called during the controller's initialization phase (such as `$onInit`).

This stage uses the Chain of Responsibility pattern as the strategy for tracing trigger points, recursively tracing call relationships layer by layer to identify the actual triggering events. Common trigger handlers include ngClickHandler, ngChangeHandler, and initHandler.

Below are introductions to three common types of trigger handlers:

1. **ngClickHandler (`ng-click`)**：These events are directly triggered by user interactions. For example, in the button below, clicking the button calls the controller's `onCancel()` method:

```
<button type="button" ng-click="onCancel()">
    {{"CANCEL"}}
</button>
```

   This type of trigger typically corresponds to button clicks, mouse events, and similar interactions.

2. **ngChangeHandler (`ng-change`)**：Used to listen for changes in user input. For example, in the following `textarea` element, the `onChange(data)` method is triggered whenever the data changes:

doi:10.6342/NTU202503567

```
<textarea
   ng-model="data"
   ng-change="onChange(data)"
>
</textarea>
```

This type of trigger is typically related to two-way binding and can be used to detect input behavior and perform corresponding actions.

3. **initHandler (`init()`)**：These methods are executed during the controller's initialization phase by internal logic and are unrelated to the template. For example, they may be invoked immediately when the controller initializes:

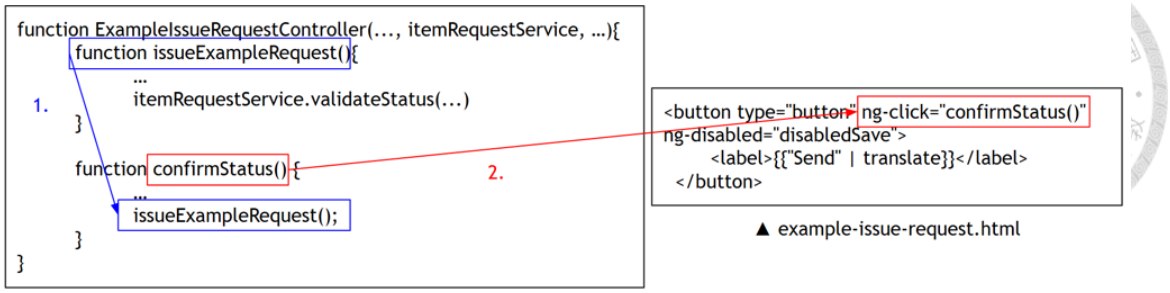
```
function ExampleApplyResultsController(...) {
    init();
    ...
    function init() {
        ...
    }
}
```

These methods are commonly used for initializing default states or loading default data and are part of the system's automatic triggering process.

By identifying the above trigger types, we can help establish the call dependency relationships from UI components to controller methods, which aids in fully reconstructing the interaction flows and behavioral logic within the frontend system.

The following example uses ngClickHandler to demonstrate the dependency relationship from UI components to controller methods:

doi:10.6342/NTU202503567



▲ xxx.js

▲ example-issue-request.html

Based on the `issueExampleRequest()` method in the controller, code analysis reveals that this method is called by `confirmStatus()`. Further tracing of its trigger points shows that `confirmStatus()` is bound to the `ng-click` attribute in the template. From this, we understand that the actual trigger event for `issueExampleRequest()` is the user clicking the ”Send” button on the interface.

At this point, the Dependency Path Map expands as follows:

```
Dependency Path Map:
Path1-1-1-1: (Type: HTTP Request, dependency: itemRequestService
  ↪ .sendRequest)
        → (Type: HTTP Request, dependency: itemRequestService.
          ↪ validateStatus)
        → (Type: controller function, dependency:
          ↪ issueExampleRequest
          <controller: ExampleIssueRequestController>)
        → (Type: htmlFile, dependency: example-issue-request.
          ↪ html)
        → (Type: ngClickHandler, dependency: <button>...</
          ↪ button> <example-issue-request.html>)
 ...
```


doi:10.6342/NTU202503567

6.6 AngularJS Routing Handlers

This process aims to start from the template and, through static analysis techniques, gradually trace back and locate the corresponding AngularJS routing state entry point.

The approach employs the Chain of Responsibility pattern, composed of multiple handlers (such as uibModalOpenRoutingHandler, HtmlRoutingHandler, etc.) connected sequentially to trace and identify the routing state entry points that lead to the current template. This method effectively reconstructs the correspondence between user interactions and system routing states, providing a foundation for subsequent navigation analysis. Below, two common handler approaches are introduced:

1. **uibModalOpenRoutingHandler**：This handler is responsible for handling modals dynamically opened via `$uibModal.open`. It analyzes the configured `templateUrl` and `controller` to locate the corresponding template and controller, thereby reconstructing the entry point and structure of such dynamic routing triggers. An example is shown below:

```
$uibModal.open({
  templateUrl: 'example-issue-request.html',
  controller: ExampleIssueRequestController,
  ...
});
```

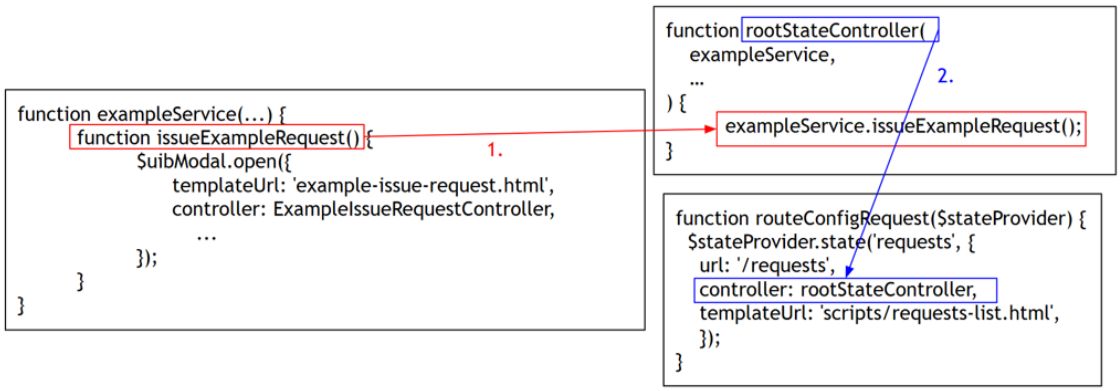
2. **HtmlRoutingHandler**：This handler analyzes whether a static template includes other custom tags, such as the custom component `<example-request-detail>` nested within the template file. By examining the nesting relationships of custom tags, it identifies the correspondence between template files and routing states,

doi:10.6342/NTU202503567

thereby tracing back to the entry point of the routing state. The following example is taken from `requests-list.html`:

```
<div>
  <example-request-detail></example-request-detail>
</div>
```

The following example uses uibModalOpenRoutingHandler to demonstrate its processing logic:



Starting from the template file `example-issue-request.html`, by analyzing where it is loaded, we find that this template is dynamically loaded via `$uibModal.open` by the `issueExampleRequest()` method in `exampleService`. Further tracing back to the controller that calls this service method, we locate `rootStateController` and identify the view `requests-list.html` bound to it.

Finally, based on the `$stateProvider.state` definitions in the routing configuration, we confirm that `requests-list.html` belongs to the state named `requests`, establishing the entry point of the state and completing the process of tracing from the template back to the state entry point.

doi:10.6342/NTU202503567

At this point, the Dependency Path Map expands as follows:

```
Dependency Path Map:
Path1-1-1-1-1: (Type: HTTP Request, dependency:
  ↪ itemRequestService.sendRequest)
          → (Type: HTTP Request, dependency: itemRequestService.
            ↪ validateStatus)
          → (Type: controller function, dependency:
            ↪ issueExampleRequest
            <controller: ExampleIssueRequestController>)
          → (Type: htmlFile, dependency: example-issue-request.
            ↪ html)
          → (Type: ngClickHandler, dependency: <button>...</
            ↪ button> <example-issue-request.html>)
          → (Type: uibModalOpenRoutingHandler, dependency:
            ↪ example-issue-request.html -> requests-list.html (
            ↪ state 'requests'))
 ...
```

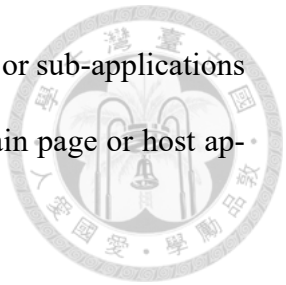
6.7 Angular Routing Pipes

This process aims to start from the identified AngularJS state definitions and further trace back to the entry point of the entire application. By analyzing the URLs corresponding to the states, routing rules, and external embedding methods (such as `<iframe>`), the starting locations where users enter the system can be determined.

This process covers the following tracking aspects:

- **URL Resolution**: Map the URL configurations defined in `$stateProvider` to actual web paths (e.g., `/requests`).

doi:10.6342/NTU202503567



- **Inline Frame Element (iframe)**: Analyze whether subsystems or sub-applications are loaded via `<iframe>` elements, and trace upward to the main page or host application.

- **Angular Routing Configuration**: In Angular routing setup, trace upward through the relationships between components and modules to identify the routing structure of specific components and locate the root module of the entire project. In AngularJS, application initialization is usually done via the `ng-app` directive or the `angular.bootstrap()` function. In Angular, the application is started by loading the main module through `platformBrowserDynamic().bootstrapModule(...)`.

This process enables the reconstruction of the entire system's navigation context, which is especially crucial for use case analysis in cross-module integration and large-scale applications. The diagram below illustrates the workflow of Angular Routing Pipes:

doi:10.6342/NTU202503567

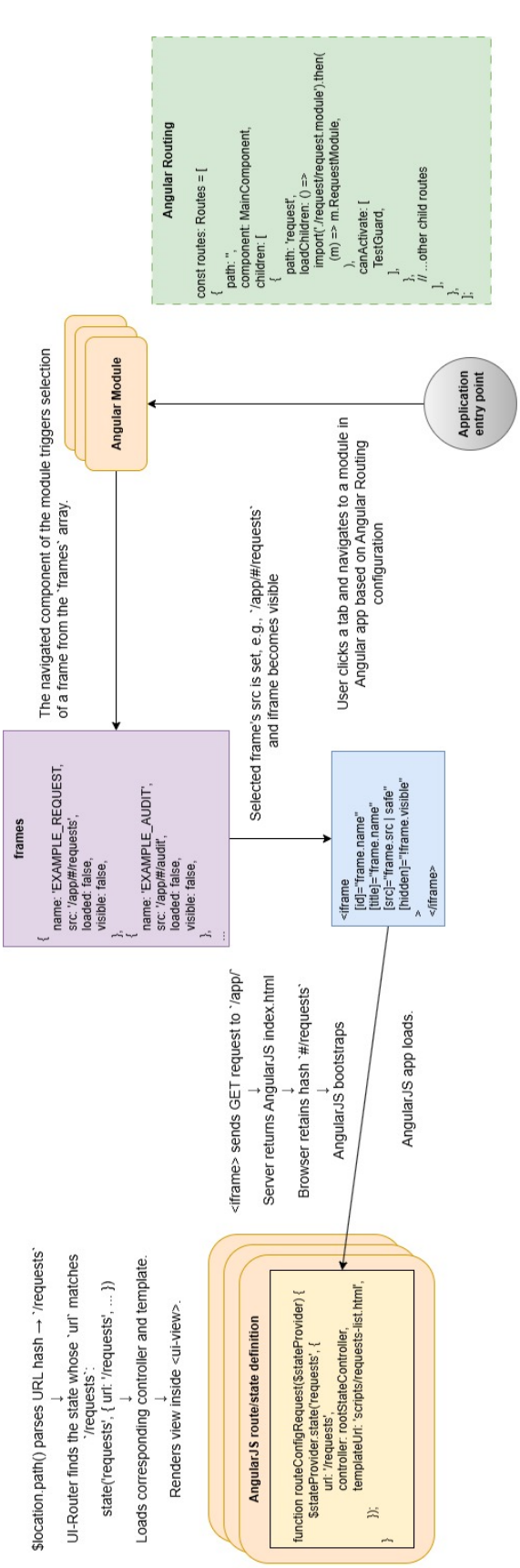
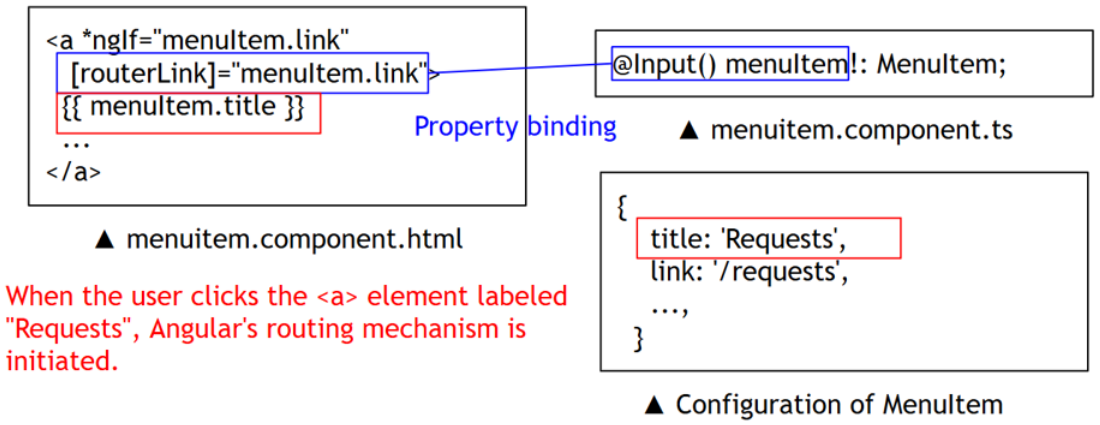


Figure 6.2: The workflow of the Angular Routing Pipes

doi:10.6342/NTU202503567

6.8 Find Routing Trigger Point Pipe

This step involves identifying HTML elements that link user interactions to routing navigation through the use of the [routerLink] attribute or other similar bindings. By recognizing these elements, we can pinpoint the exact locations and methods by which users trigger route changes in the interface.



The final Dependency Path Map is as follows:

```
Dependency Path Map:
Path1-1-1-1-1-1-1: (Type: HTTP Request, dependency:
  ↪ itemRequestService.sendRequest)
        → (Type: HTTP Request, dependency: itemRequestService.
          ↪ validateStatus)
        → (Type: controller function, dependency:
          ↪ issueExampleRequest
          <controller: ExampleIssueRequestController>)
        → (Type: htmlFile, dependency: example-issue-request.
          ↪ html)
        → (Type: ngClickHandler, dependency: <button>...</
```


doi:10.6342/NTU202503567

```
  ↪ button> <example-issue-request.html>)
→ (Type: uibModalOpenRoutingHandler, dependency:
  ↪ example-issue-request.html -> requests-list.html (
  ↪ state 'requests'))
→ (Type: Angular Routing,
   dependency: AngularJS route/state definition ->
     ↪ iframe src -> Angular Module/Component ->
     ↪ Angular Routing Path)
→  (Type: Routing Trigger Point, dependency: <a
  ↪              *ngIf="menuItem.link" [routerLink]="
  ↪ menuItem.link"> {{ menuItem.title }}...</a> (<a>:
  ↪ user clicks a tab))
```

6.9 Build Use Case Model Pipe

Based on the Dependency Path Map analyzed in the previous pipes, the use case model is constructed using the Builder Pattern. This approach systematically combines various dependency elements to progressively build a complete and structured use case model.

6.9.1 Use Case Template

The generated use case model will serve as input for a subsequent Machine Learning model, which is responsible for automatically transforming the use case model into a task model [13]. Therefore, the use case specification template we employ is as follows:

doi:10.6342/NTU202503567

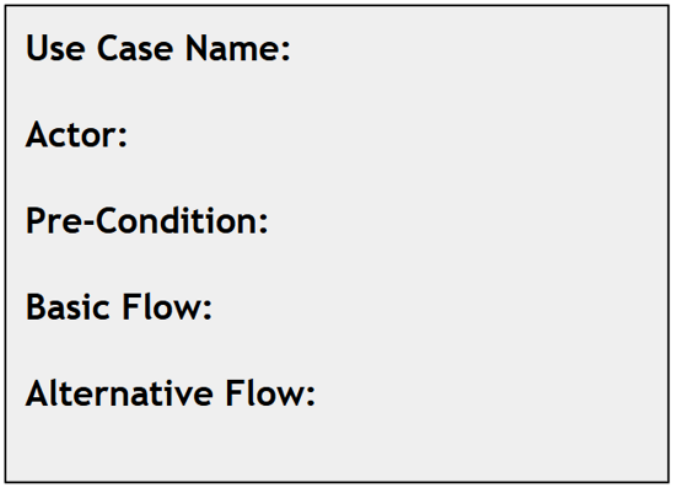


Figure 6.3: Use Case Specification Template

The use case specification template includes the following five main fields:

1. **Use Case Name**: Describes the functionality or behavior represented by the use case, serving as a clear functional unit within the overall requirements analysis.

2. **Actor**: The role(s) involved in the use case.

3. **Pre-Condition**: Defines the conditions that must be satisfied before the use case begins to ensure appropriate prerequisites for its execution. In the Use Case Specification, Pre-Condition can also represent dependencies among use cases, enabling the derivation of the entire system's Use Case Diagram.

4. **Basic Flow**: Details the primary successful path of the use case, i.e., the typical interaction process between the user and the system under normal circumstances.

5. **Alternative Flow**: Describes possible process branches and system responses in abnormal or special situations.

doi:10.6342/NTU202503567

6.9.2 Steps for Building Use Case Model

This subsection outlines the steps to transform dependencies into a use case model.

Step 1. Generate Use Case Specifications from the dependencies collected through the pipes spanning from FindHttpRequestPipe to HtmlTriggerHandlers.

Step 2. When encountering HTML routing file information recorded in AngularJSRouting-Handlers, generate a Use Case Specification for each HTML file, indicating that the user has entered the corresponding view.

Step 3. For each Use Case Specification, collect the following information:

- HTML trigger information recorded in HtmlTriggerHandlers,
- HTML file information recorded in FindHtmlHandlers,
- HTML routing file information recorded in AngularJSRoutingHandlers.

Step 4. Generate the pre-condition for the Use Case Specification.

- **Case 1**: If the HTML file recorded in FindHtmlHandlers is a view used in a $stateProvider state, then the pre-condition dependencies originate from AngularRoutingPipes and FindRoutingTriggerPointPipe.
- **Case 2**: If the HTML file recorded in FindHtmlHandlers is **not** a view used in any $stateProvider state, then the pre-condition originates from the HTML routing file information in AngularJSRoutingHandlers.

doi:10.6342/NTU202503567

6.9.3 Pipes, Dependencies and Use Case Mapping

The following table shows the correspondence between dependencies and their pipes, and the components of the use case. Among them, the generation of the "Use Case Name" requires assistance from natural language processing techniques (such as spaCy [14]).

Table 6.1: Correspondence between Dependencies, Pipes, and Use Case Elements

Pipe	Dependency	Use Case Name	Actor	Pre-Condition	Basic Flow	Alternative Flow
Find Http Request Pipe	HTTP Request	v (Extract noun and verb from the function name containing the HTTP request using spaCy)	v (Backend verifies the request)		v	v
FindController Pipe	controller function	v (Method name of controller, spaCy)				
FindHtmlHandlers	htmlFile	v		v	v	

Pipe	Dependency	Use Case Name	Actor	Pre-Condition	Basic Flow	Alternative Flow
Html Trigger Handlers	ngClickHandler	v (Button name, etc)			v	v (cancel button beside)
	ngChangeHandler	v (ng-model)			v	v (input null)
	initHandler	v (The name of fields displays to user)			v	
AngularJS Routing Handlers	uibModalOpenRoutingHandler			v		
	HtmlRoutingHandler			v		

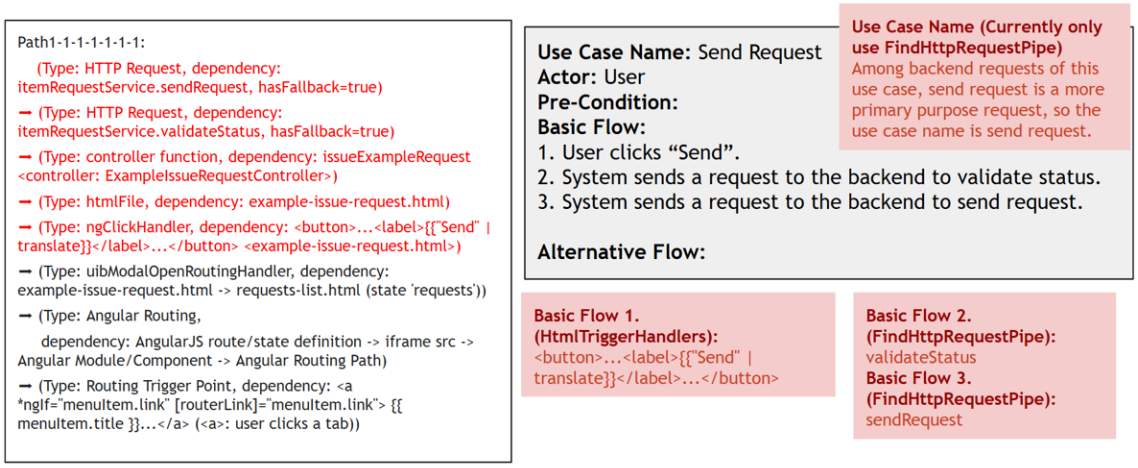
Pipe	Dependency	Use Case Name	Actor	Pre-Condition	Basic Flow	Alternative Flow
Angular Routing Pipes	Iframe,			v		
	Angular Component/Module,			v		
	Angular Routing Path		v (Guard)	v		
Find Routing Trigger Point Pipe	Native html element			v		

doi:10.6342/NTU202503567

6.9.4 Build Use Case Model Example

This subsection demonstrates how to generate use case specifications from the dependency path based on the steps outlined in Section 6.9.2.

Step 1: Generate Use Case Specifications from the dependencies collected through the pipes spanning from FindHttpRequestPipe to HtmlTriggerHandlers.

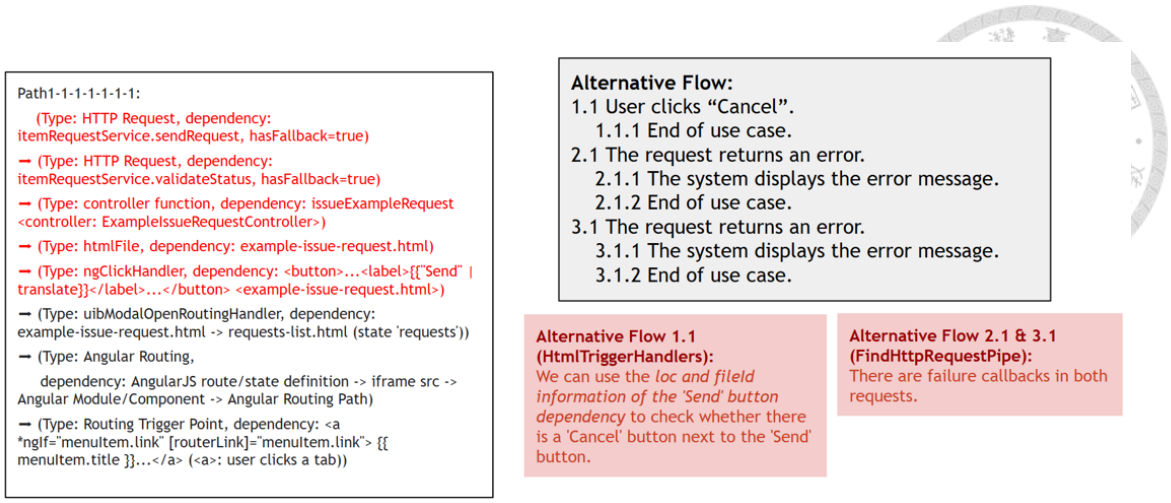


Use Case Name is derived from the backend requests found by FindHttpRequestPipe, specifically those with a primary purpose. In this use case, `sendRequest` is the request with the main task objective, thus named **Send Request**.

Basic Flow 1 is derived from the HTML element trigger information captured by HtmlTriggerHandlers (e.g., `<button>...<label>{{"Send" | translate}}</label> ...</button>`), representing the user clicking the "Send" button.

Basic Flows 2 and 3 correspond to two backend requests analyzed by FindHttpRequestPipe, namely `validateStatus` and `sendRequest`, which represent the system's basic process executed after the user clicks the button.

doi:10.6342/NTU202503567



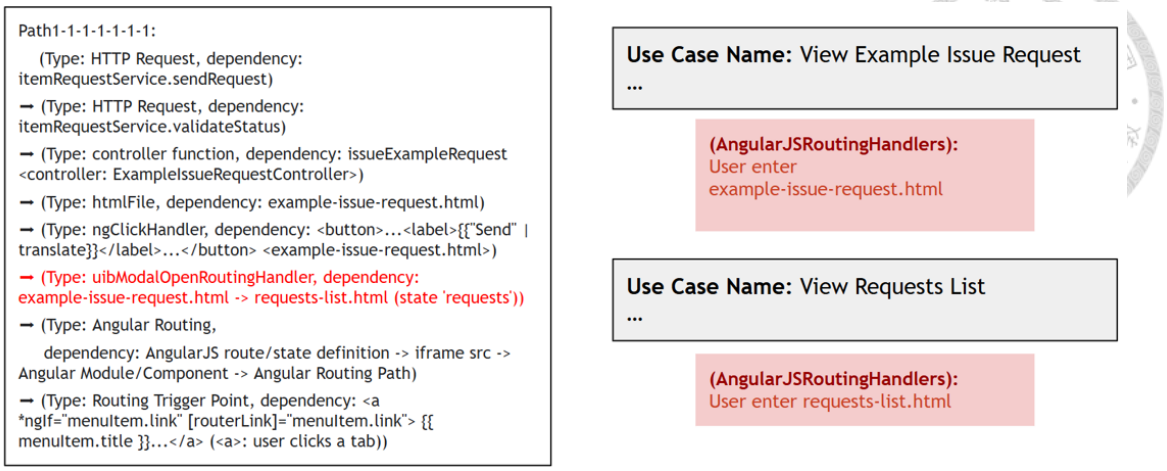
Alternative Flow 1.1 is based on the `loc` and `fileId` information of the `Send` button's dependency captured by HtmlTriggerHandlers, used to determine whether a `Cancel` button exists next to the `Send` button.

Alternative Flow 2.1 and 3.1 correspond to two backend requests analyzed by FindHttpRequestPipe, both containing failure callback functions, representing the handling processes when the requests fail.

Step 2: When encountering HTML routing file information recorded in AngularJSRoutingHandlers, generate a Use Case Specification for each HTML file, indicating the user has entered the corresponding view.

For example, in this dependency path, when routing information reaches `example-issue-request.html`, it indicates the user has entered this view, and a Use Case Specification is generated for this HTML file. Similarly, when routing information reaches `requests-list.html`, it indicates the user has entered the corresponding view, and a corresponding Use Case Specification is also generated.

doi:10.6342/NTU202503567

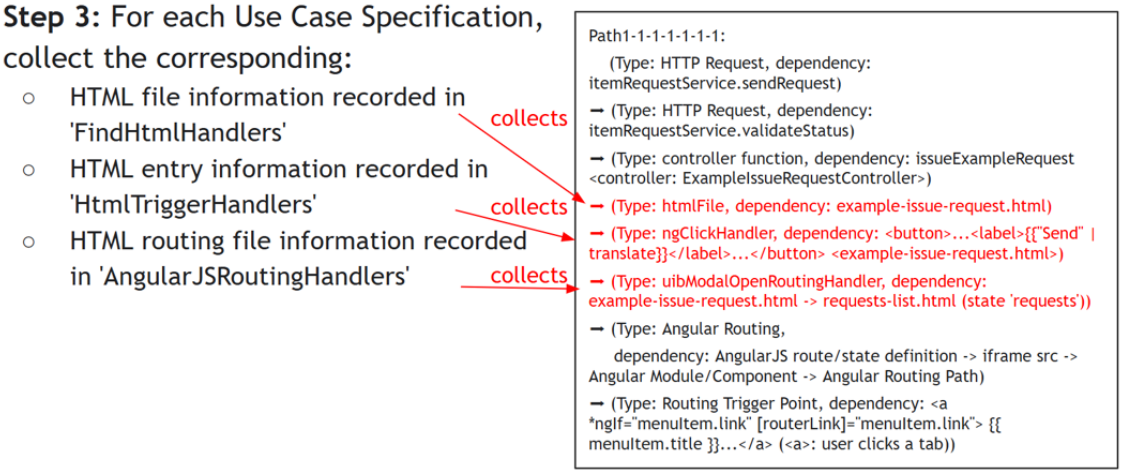


Step 3: For each Use Case Specification, collect the following information as the basis for generating Pre-Conditions in Step 4:

(1) **HTML trigger information** recorded in **HtmlTriggerHandlers**,

(2) **HTML file information** recorded in **FindHtmlHandlers**,

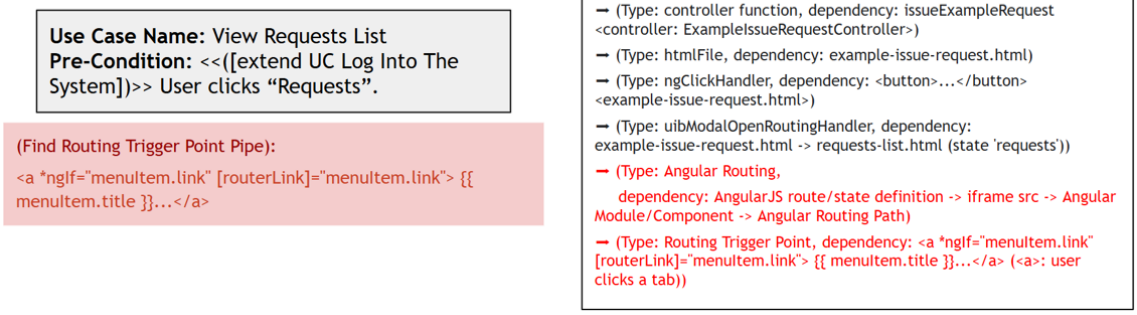
(3) **HTML routing file** information recorded in **AngularJSRoutingHandlers**.



Step 4：Generate the pre-condition for the Use Case Specification.

doi:10.6342/NTU202503567

- **Case 1**: If the HTML file recorded in FindHtmlHandlers is a view used in a `$stateProvider` state, then the pre-condition dependencies originate from AngularRoutingPipes and FindRoutingTriggerPointPipe.



The use case **View Requests List** occurs in `requests-list.html`, which is a view used by a certain $stateProvider state. Therefore, its pre-condition is based on AngularRoutingPipes and FindRoutingTriggerPointPipe.

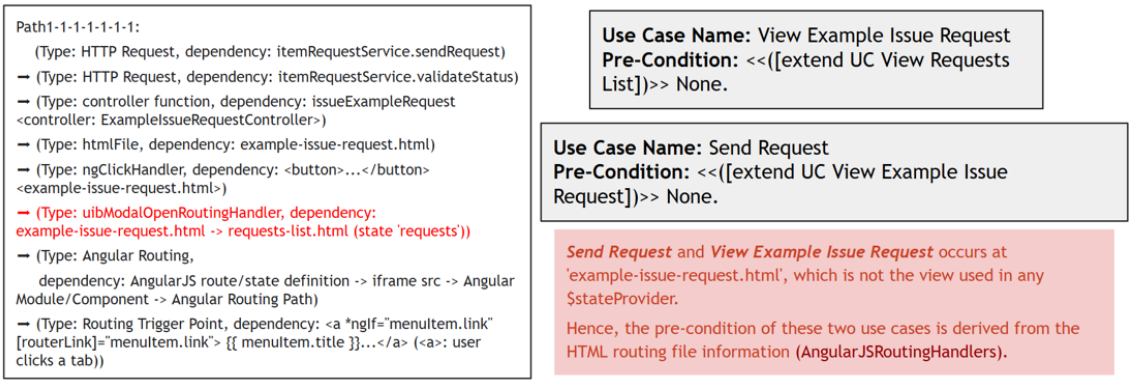
FindRoutingTriggerPointPipe captures the HTML elements that trigger routing, for example:

```
<a *ngIf="menuItem.link" [routerLink]="menuItem.link">
    {{ menuItem.title }}...
</a>
```

This means when the user clicks the <a> element, routing to the view of the current use case is triggered. Therefore, this action (User clicks "Requests") is considered the pre-condition of the use case.

doi:10.6342/NTU202503567

- **Case 2**: If the HTML file recorded in FindHtmlHandlers is **not** a view used in any `$stateProvider` state, then the pre-condition originates from the HTML routing file information in AngularJSRoutingHandlers.



Use cases **Send Request** and **View Example Issue Request** occur in `example-issue-request.html`, which is not used as a view in any `$stateProvider` state. Therefore, their pre-conditions are derived from the HTML routing information recorded in AngularJSRoutingHandlers.

The resulting use case specification is as follows:

Use Case Name: Send Request
Actor: User
Pre-Condition: <<([extend UC View Example Issue Request])>> None.
Basic Flow:
1. User clicks “Send”.
2. System sends a request to the backend to validate status.
3. System sends a request to the backend to send request.

Alternative Flow:
1.1 User clicks “Cancel”.
  1.1.1 End of use case.
2.1 The request returns an error.
  2.1.1 The system displays the error message.
  2.1.2 End of use case.
3.1 The request returns an error.
  3.1.1 The system displays the error message.
  3.1.2 End of use case.

Figure 6.4: Generated Use Case Specification

doi:10.6342/NTU202503567

Chapter 7 Conclusion

This study adopts reverse engineering techniques to parse Angular and AngularJS frontend code into abstract syntax trees, which are then converted into structured data and stored in a relational database. Based on this data, further analysis is conducted to reconstruct the system's use case model. The process preserves the original semantics and structure of the source code, providing a solid foundation for subsequent static analysis and dependency exploration.

By leveraging the AST node information stored in the database, this research enables multi-level analysis across modules, controllers, HTTP requests, and user interface components. The system employs a bottom-up semantic tracing approach to automatically identify user interactions and system responses, thereby restoring dynamic usage scenarios.

Ultimately, through AST node analysis, this study automates the generation of use case models, helping users quickly understand system functions and interaction flows. This facilitates the linkage between source code and high-level system behavior, enhancing the efficiency and accuracy of software system analysis and maintenance.

doi:10.6342/NTU202503567

Chapter 8 Future Work

1. Since the use case model is reverse-engineered from the source code, a semantic gap still exists. In the future, verb classification techniques will be introduced to help users quickly identify the primary action of each use case.

2. Currently, the tracing of call expressions cannot accurately determine the origin of arguments sent to the backend. Future work will adopt data flow analysis to trace the entire propagation of argument values from user interface events to HTTP requests.

3. Role identification heavily depends on backend logic and persistent data. Future work will integrate backend tracing and database interactions to improve the accuracy of role identification.

4. The identification of alternative flows still needs refinement to ensure that all possible behavioral paths of the system are fully captured.

5. The automatically generated use case models need to be systematically validated against the original project to ensure their completeness and correctness.

6. Future work will incorporate the identification of use cases that are handled entirely on the frontend without the need to send HTTP requests to the backend. For example, `window.open(url)` opens a specified URL in a new window without sending a request to the backend, thus representing a frontend-handled use case.

doi:10.6342/NTU202503567

7. Currently, all generated use case specifications are modeled using the *extend* relationship. Future work will explore how to identify *include* relationships from frontend source code. For example, overlapping segments in dependency paths can serve as evidence for *include* relationship.

8. When a single use case specification includes multiple HTTP requests, further analysis is needed to determine which request best represents the primary behavior of the use case.

doi:10.6342/NTU202503567

References

[1] Enrique A Miranda, Mario Berón, Germán Montejano, and Daniel Riesco. Using reverse engineering techniques to infer a system use case model. Journal of Software: Evolution and Process, 31(2):e2121, 2019.

[2] Gerardo Canfora and Massimiliano Di Penta. New frontiers of reverse engineering. In Future of Software Engineering (FOSE'07), pages 326–341. IEEE, 2007.

[3] Ravi Khadka, Amir Saeidi, Slinger Jansen, and Jurriaan Hage. A structured legacy to soa migration process and its evaluation in practice. In 2013 IEEE 7th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems, pages 2–11. IEEE, 2013.

[4] Ravi Khadka, Amir Saeidi, Slinger Jansen, Jurriaan Hage, and Geer P Haas. Migrating a large scale legacy application to soa: Challenges and lessons learned. In 2013 20th working conference on reverse engineering (WCRE), pages 425–432. IEEE, 2013.

[5] Elliot J. Chikofsky and James H Cross. Reverse engineering and design recovery: A taxonomy. IEEE software, 7(1):13–17, 2002.

[6] Angularjs. https://angularjs.org/.

doi:10.6342/NTU202503567

[7] Angular. https://angular.dev/.

[8] parse5. https://www.npmjs.com/package/parse5.

[9] @babel/parser. https://babeljs.io/docs/babel-parser.

[10] @typescript-eslint/ parser. https://www.npmjs.com/package/@typescript-eslint/parser.

[11] Babel. https://babeljs.io/.

[12] Eslint. https://eslint.org/.

[13] Y.-Y. Chung. Auto-transform use case specifications to task models using machine learning model. Master's thesis, National Taiwan University, 2024.

[14] Matthew Honnibal. spacy 2: Natural language understanding with bloom embeddings, convolutional neural networks and incremental parsing. (No Title), 2017.

doi:10.6342/NTU202503567