國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science & Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

從任務模型自動建構使用者介面

Auto Build User Interface from Task Model

劉仁軒

LIOU, REN-SHIUAN

指導教授：李允中 博士

Advisor: Jonathan Lee, Ph.D.

中華民國 112 年 7 月

July, 2023

doi:10.6342/NTU202303645

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

從任務模型自動建構使用者介面

Auto Build User Interface from Task Model

本論文係劉仁軒君（學號 R10922151）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 112 年 7 月 27 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 Department of Computer Science and Information Engineering on 27 July 2023 have examined a Master's thesis entitled above presented by LIOU, REN-SHIUAN (student ID: R10922151) candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

（指導教授 Advisor）

系主任/所長 Director:

誌謝

首先我要感謝我的指導教授李允中在過去兩年間對我的仔細教導以及研究方法的傳授：如何找到研究主題，並有系統地拆解與分析問題，進而提出解決方案。這使我得以更準確地理解並掌握問題的切入點，不斷在研究的道路上成長進步。除此之外，我也衷心感謝台灣大學軟體工程實驗室的所有成員，特別感謝許恆、陳力聖、林辰臻、林怡伶、張馨尹、梁峻瑞、錢怡君與學弟妹們的悉心協助。他們的合作與討論，以及彼此之間的互助，使這篇論文得以順利完成。

doi:10.6342/NTU202303645

摘要

在軟體開發的過程中，將需求轉化爲可運行的程式需要經歷一連串具體化的步驟。本篇論文的目的在於設計一套自動化的機制，能夠逐步將軟體需求書中的使用案例轉化爲使用者介面，並在轉化的過程中將生成的使用者介面元件與後端的服務元件進行綁定。

爲了實現這個目標，我們引入了任務模型作爲中間產物，來表達使用者在運用軟體時所需完成的任務以達成目標。這些任務資訊被用來生成使用者介面描述語言，從而能夠自動化地生成使用者介面。同時，我們還進一步增強了使用者介面描述語言的表達能力，以便生成更現代化的使用者介面。

關鍵詞 — 使用者介面元件、使用者介面描述語言、服務元件、任務模型、使用案例

doi:10.6342/NTU202303645

Abstracts

In the software development process, transforming requirements into user interface involves a series of steps. The main purpose of this thesis is to design an automated process that translates the use cases from the software requirements document(SRS) into user interfaces, and binds the UI components with service components during the transformation.

To achieve this goal, we introduce task model to express the tasks users need to accomplish to achieve their goals when using the software. This information is used to generate a user interface description language, which can automatically transform into user interfaces. Additionally, we further enhance the expressive power of the user interface description language to create more modern user interfaces.

Index terms — UI Component, UI Description Language, Service Component, Task Model, Use Case

doi:10.6342/NTU202303645

Contents

口試委員審定書 **i**

誌謝 **ii**

摘要 **iii**

Abstracts **iv**

List of Figures **viii**

List of Tables **xi**

Chapter 1 Introduction **1**

Chapter 2 Related Work **4**

2.1 Cameleon Reference Framework 4

2.2 UI Component 5

2.3 User Interface Description Languages 6

2.4 ConcurTaskTree 10

doi:10.6342/NTU202303645

2.5 Dijkstra's Two-Stack Algorithm 13

Chapter 3 UI Component's Conditional Behavior 15

3.1 Categorize Angular APIs 16

3.2 Define Variables in UI Description Language 18

3.3 Design and Implementation 19

Chapter 4 Modeling Tasks 22

4.1 Task Type Defining 22

4.2 ConcurTaskTrees Modeling 25

Chapter 5 UIDL Generating Algorithm 28

5.1 Use Case to Task Model 29

5.1.1 Use Case Diagram to Main CTT 30

5.1.2 Use Case Spec to Use Case CTT 34

5.2 CTT to UIDL Mapping 37

5.2.1 Retrieve UI Component and Binding Information 38

5.2.2 Determine Navigation of the UI 42

5.2.3 Design on Task Converter 49

Chapter 6 Conclusion 53

6.1 Summary 53

6.2 Future work 54

doi:10.6342/NTU202303645

Bibliography **55**

doi:10.6342/NTU202303645

List of Figures

2.1 Page Description Language 8

2.2 Navigation Description Language 9

2.3 Service UI Mapping Description Language 9

2.4 CTT Hierarchical structure 10

2.5 CTT Task allocation 11

3.1 Idea of Variable 19

3.2 Overall UI Component Decorator 20

4.1 CTT Modeling 26

4.2 Actor's Hierarchical Structure 27

4.3 Specify Actors within Tasks 27

5.1 UI Generating System Architecture 28

5.2 Pattern of Main CTT 31

5.3 Use Cases Enabled by t_A 33

5.4 Rule 1 for Constructing CTT 36

doi:10.6342/NTU202303645

5.5 Rule 2 for Constructing CTT 36

5.6 Example of Constructing CTT by ID 37

5.7 Example of CTT to UI Component Mapping(1): Main Tree 40

5.8 Example of CTT to UI Component Mapping(2): Register 40

5.9 Example of CTT to UI Component Mapping(3): Back to Main Tree 40

5.10 Example of CTT to UI Component Mapping(4): View List of Item . 41

5.11 Example of CTT to UI Component Mapping(5): Back to Main Tree Again 41

5.12 Example of CTT to UI Component Mapping: Final Product 41

5.13 Example of Determine Navigation(1): Main Tree(1) 45

5.14 Example of Determine Navigation(2): Register 46

5.15 Example of Determine Navigation(3): Register 46

5.16 Example of Determine Navigation(4): Register(Pop Operator Stack) . 46

5.17 Example of Determine Navigation(5): Register 47

5.18 Example of Determine Navigation(6): Main Tree 47

5.19 Example of Determine Navigation(7): Main Tree 47

5.20 Example of Determine Navigation(8): Main Tree 48

5.21 Example of Determine Navigation(9): Main Tree(Pop Operator Stack) 48

5.22 Example of Determine Navigation: Final Product 48

5.23 Visitor Pattern on CTT 49

5.24 Builder Pattern for UIDL 51

doi:10.6342/NTU202303645

5.25 Task Converter 52

doi:10.6342/NTU202303645

List of Tables

2.1 Temporal Operator of CTT 12

5.1 Information in Each Task Type 39

5.2 Information in Temporal Operators 43

5.3 Behaviors when Pushing/Popping Operators 44

doi:10.6342/NTU202303645

Chapter 1

Introduction

In the process of developing frontend, the transformation from software requirements to user interface usually involves multiple steps. These steps include establishing requirements based on user descriptions, designers creating desired appearance based on the requirements, and then programmer is responsible for writing the code to give the user interface with corresponding dynamic behaviors. Numerous discussions and modifications are carried out throughout the process to complete the development.

However, due to the hierarchical communication structure, issues related to efficiency and accuracy may arise. Every step of the development process involves different team members, and there may be multiple factors that can make communication complex and time-consuming. This can lead to inefficiency and delays in the development progress, and may even cause the final outcome different from the original requirements.

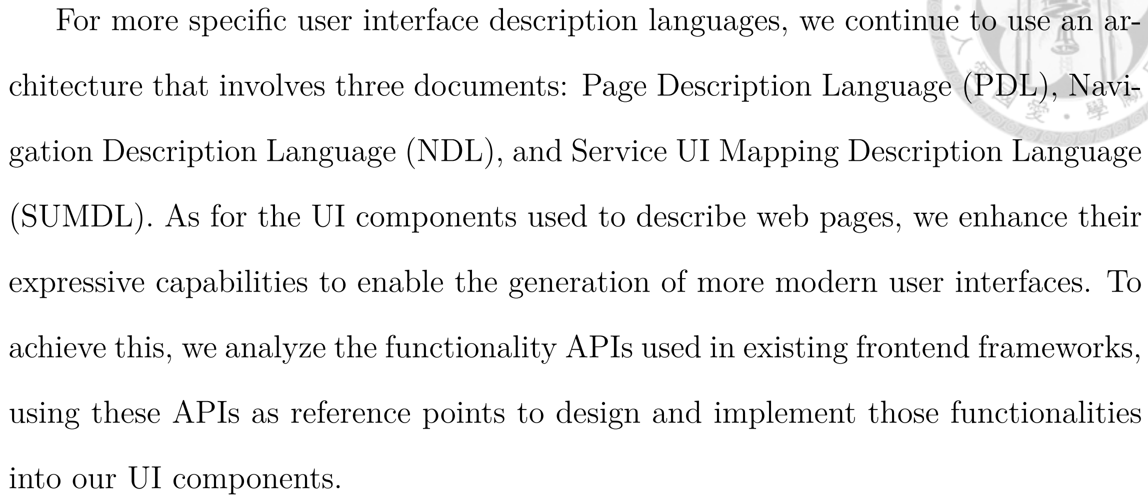
doi:10.6342/NTU202303645

Therefore, the goal of this research is to propose a User Interface Description Language (UIDL) that is capable of expressing multiple abstraction levels. To achieve this, a task model will be introduced as one of these abstraction layers, and algorithms for converting between different abstraction levels will be designed and implemented.

This provides us with several advantages. Firstly, a higher level of abstraction in the description language, compared to lower levels, is closer to natural language. This enables project members to express themselves more clearly during communication, thereby enhancing communication efficiency and accuracy while aligning with software requirements. With a description language written at a high abstraction level, the development team is able to comprehend and share their thoughts on the requirements and design, reducing misunderstandings or mistakes that might arise from communication. As a result, team collaboration efficiency is improved.

Secondly, designing and implementing transformation algorithms can reduce the requirement of programming ability, which helps reduce the costs and shorten the development timeline. In traditional development processes, developers is required to have a certain level of programming expertise and invest a significant amount of time in writing complex code to create the corresponding user interfaces. However, the transformation algorithms provided by this research can automatically convert highly abstract description languages into code, allowing developers to focus on requirement design without needing to dedicate time and effort to the implementation details. This not only speed up the development process but also enhances the efficiency and productivity of the development team.

doi:10.6342/NTU202303645



For more specific user interface description languages, we continue to use an architecture that involves three documents: Page Description Language (PDL), Navigation Description Language (NDL), and Service UI Mapping Description Language (SUMDL). As for the UI components used to describe web pages, we enhance their expressive capabilities to enable the generation of more modern user interfaces. To achieve this, we analyze the functionality APIs used in existing frontend frameworks, using these APIs as reference points to design and implement those functionalities into our UI components.

Therefore, the rest of the paper is organized as follows: In Chapter Two, we provide an introduction to the related work, including the abstraction layer defined by the Cameleon Reference Framework, UI components, other UI description languages, the notation for task model specifications called ConcurTaskTree (CTT), and Dijkstra's Two-Stack Algorithm, which we will employ in our transforming process. In Chapter Three, we will delve into the details of UI components and how we have augmented them. In Chapter Four, the task model will be defined and modeled. In Chapter Five, we will cover the process of transforming from use cases to the task model, as well as the algorithms for transitioning from the task model to the UI description language.

doi:10.6342/NTU202303645

Chapter 2

Related Work

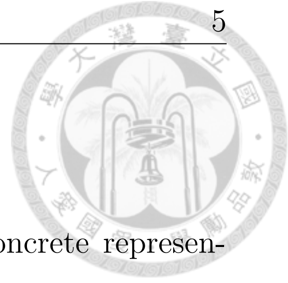
2.1 Cameleon Reference Framework

The Cameleon Reference Framework [3] aims to provide a framework for analyzing user interfaces that support multiple contexts of use. It divides the user interface development lifecycle into four levels, each defining user interfaces at different levels of abstraction. The following are the user interfaces defined at each abstraction level:

1. Task & Concepts: The most abstract level of describing user interfaces, defining various interactive tasks that users can perform and the domain objects affected by these tasks.

2. Abstract UI: It defines user interfaces independently from the interactors available on the targets, including graphical and voice interactions. This definition treats the user interface as a collection of workspaces and then specifies the

doi:10.6342/NTU202303645



interactive relationships between these workspaces.

3. Concrete UI: This level transforms the Abstract UI into concrete representations related to the interactors, describing navigation mechanisms within the user interface. The definitions at this level are translated into actual UI appearance that directly interact with users.

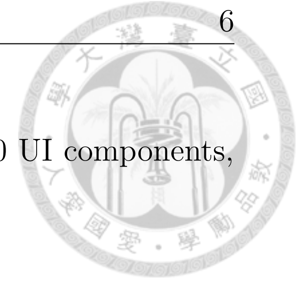
4. Final UI: This represents the executable user interface. It might be influenced by platforms, devices, or other constraints, and thus can only run in specific software and hardware environments.

2.2 UI Component

UI component is the fundamental interface component that users can interact with and visualize in web applications. UI Components usually fall into one or more of the following four categories[7]:

- Input control: UI components that allow users to enter information into the system.
- Navigation: UI components that will enable users to navigate across content within the application.
- Informational: UI components that display information in various ways.
- Container: UI components that consist of other UI components.

doi:10.6342/NTU202303645



In Ming-Hsuan's thesis [4], we have defined and categorized 30 UI components, and modeled them using the following two design patterns.

- Composite pattern: To encapsulate the composite structure of UI components. It allows UI components to integrate others while still being treated the same as individual ones.

- Decorator Pattern: To add responsibilities to UI components dynamically. This design pattern allows the extension of functionalities by placing components within decorators without the need to modify the definition of the UI components.

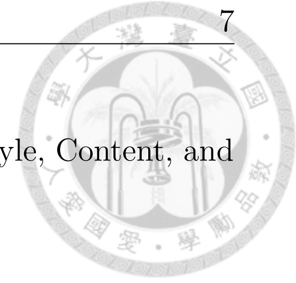
2.3 User Interface Description Languages

The goal of User Interface Description Languages (UIDL) is to define a high-level computer language for describing characteristics of a user interfaces and may be used to generate the code of the UI automatically. The following are some of the UI description languages widely used.

- User Interface Markup Language (UIML): A UIML[1] is an XML-based markup language document consisting of four major elements: Head, Interface, Peers, and Template.

  The Head element contains the metadata, which does not affect the user interface. The Peers and the template part of the UIML define the relation to other UIML documents. The Interface part describes the element that holds

doi:10.6342/NTU202303645



the information of the user interface, including Structure, Style, Content, and Behavior.

- User Interface Extensible Markup Language (UsiXML): UsiXML[5] allows designers to specify a user interface on multiple abstractions. The framework supports the transformation of each step or different contexts of use by defining a transformation model.

- Model-based Language for Interactive Applications (MARIA XML): MARIA XML[6] covers the Model, abstract UI, and concrete UI levels. It describes the functionality and which component to use but not the detailed look of the UI. Moreover, MARIA XML provides multiple Data Models to explain its behavior in response to a different event and declare the backend functions used by the user interface.

Currently, we are describing our user interface using three documents: Page Description Language (PDL), Navigation Description Language (NDL), and Service UI Mapping Description Language (SUMDL). The following is the introduction to these three documents:

- Page Description Language(Figure 2.1): PDL describes the composition of a page in a web application and the interaction of UI components. It documents the basic information of a page and the UI components that the page contains.

- Navigation Description Language(Figure 2.2): NDL describes navigation information within a page, including the information that is passed from other

doi:10.6342/NTU202303645

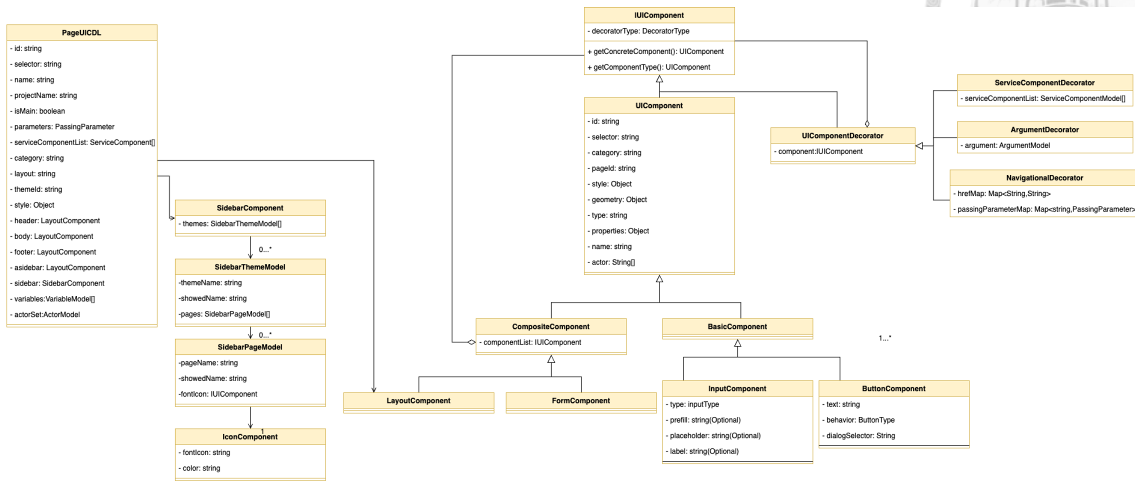


Figure 2.1: Page Description Language

pages when they navigate in (called passing parameter), and it also specifies which UI component will trigger the navigation mechanism along with the passing parameter it carries.

- Service UI Mapping Description Language(Figure 2.3): SUMDL describes the service components used in a web page. It contains information about the service component and how it is triggered, as well as how the services' return value is used.

doi:10.6342/NTU202303645

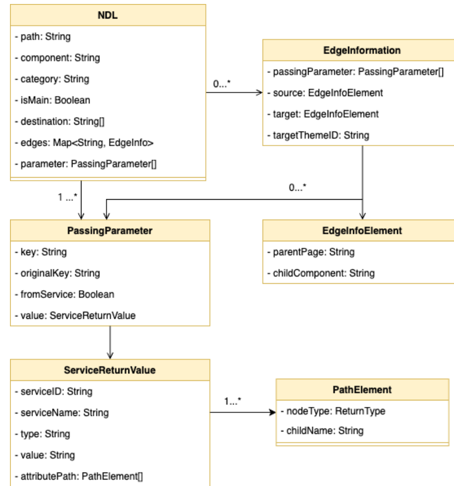


Figure 2.2: Navigation Description Language

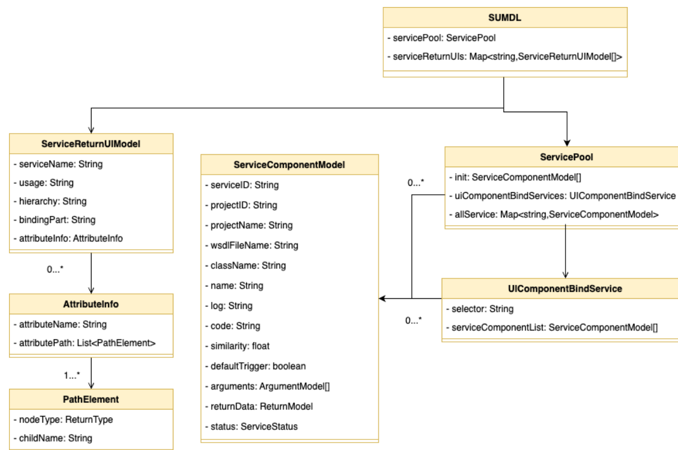


Figure 2.3: Service UI Mapping Description Language

doi:10.6342/NTU202303645

2.4 ConcurTaskTree

A task model describes how the user and system behave to achieve certain goal.

ConcurTaskTrees (CTT) is a representation of task models that supports the design of user interface. Through ConcurTaskTrees, designers can gain a clearer understanding and effectively express the activities required for users to achieve their goals. This facilitates the implementation and optimization of user interfaces.

The main features of ConcurTaskTrees are:

- Hierarchical structure: The hierarchical structure (Figure 2.4) is intuitive for human when it comes to problem-solving. It offers a mechanism to break down tasks into smaller sub-tasks for completion, resulting in a wider granularity of tasks within the task tree.

- Task allocation: CTT categorizes tasks into four categories(Figure 2.5)，
  - System tasks: Represent tasks executed by the system, including behaviors related to displaying data and performing computations.

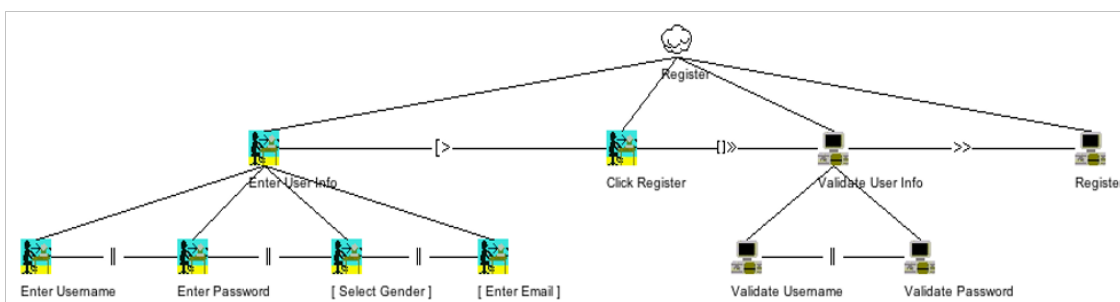


Figure 2.4: CTT Hierarchical structure

doi:10.6342/NTU202303645

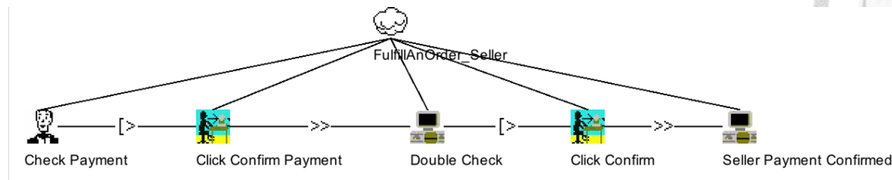


Figure 2.5: CTT Task allocation

  - Interaction tasks: Represent tasks that require users to provide information to the system.
  - User tasks: Represents the decision points on the users part, no interaction with the system.
  - Abstraction: Split up in different kinds of tasks .

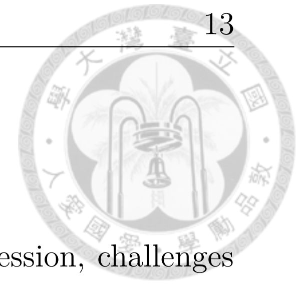
- Rich set of temporal operators: CTT introduces 8 temporal operators to define the relationships between tasks. Defining temporal relationships between tasks benefits us in UI design. Table 2.1 provides an overview of these temporal operators. We list them in descending order of priority, from the operators with the highest priority to the ones with the lowest.

doi:10.6342/NTU202303645

Table 2.1: Temporal Operator of CTT

<table>
<tr><th>Name</th><th>Notation</th><th>Definition</th></tr>
<tr><td>Choice</td><td>[]</td><td>Specifies that once one of two tasks is enabled and has started, the other task is no longer enabled.</td></tr>
<tr><td>Task Independence</td><td>|=|</td><td>Tasks can be performed in any order, but when one starts, it must finish before the other one can start.</td></tr>
<tr><td>Concurrent</td><td>||</td><td>Tasks can be performed in any order, or at the same time.</td></tr>
<tr><td>Concurrent Communicating</td><td>|[]|</td><td>Tasks that can exchange information while performed concurrently.</td></tr>
<tr><td>Disabling</td><td>[></td><td>The first task is completely interrupted by the second task.</td></tr>
<tr><td>Suspend-Resume</td><td>|></td><td>The first task can be interrupted by the second one. When the second terminates then the first one can be reactivate from the state reached before.</td></tr>
<tr><td>Enabling</td><td>>></td><td>Specifies second task cannot begin until the first task performed.</td></tr>
<tr><td>Enabling with Information Passing</td><td>[]>></td><td>Specifies second task cannot begin until the first task performed, and that information produced in first task is used as input for the second one.</td></tr>
</table>

doi:10.6342/NTU202303645



2.5 Dijkstra's Two-Stack Algorithm

If there are multiple binary operators in a mathematical expression, challenges related to operator precedence can arise. In other words, interpreting operators from left to right might lead to incorrect results. For example, consider the expression 1 + 2 * 3. If we simply evaluate from left to right, we would obtain the incorrect result of 9.

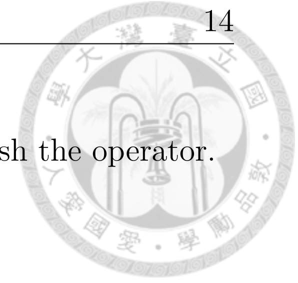
Therefore, various methods have been proposed to address the issue of operator precedence in calculations. One of these methods is Dijkstra's Two-Stack Algorithm. The concept of this algorithm lies in utilizing two stacks: one for handling operands and the other for handling operators. Its main approach is to process each character of the mathematical expression sequentially, and then perform corresponding operations on the two stacks.

For each character, perform operations based on the following rules:

- **If it is an operand**, push it onto the operand stack.
- **If it is an operator**, check if the operator at the top of the operator stack has higher precedence than the operator to be pushed:
  - If the result is negative, push the operator onto the stack.
  - If the result is positive, pop the operator from the top of the stack along with two operands from the operand stack, perform the operation, and push the result back into the operand stack.

    Repeat this process until the precedence of the operator at the top of the

doi:10.6342/NTU202303645



stack is "less than" the operator to be pushed, then push the operator.

- **If it is "("**, push it onto the operator stack.
- **If it is ")"**, repeatedly perform the operation until the operator at the top of the operator stack is "(". Then, pop the "(" from the operator stack.

doi:10.6342/NTU202303645

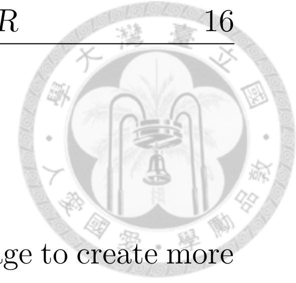
Chapter 3

UI Component's Conditional Behavior

As mentioned earlier, our approach of describing user interfaces involves reusing predefined UI components. In Ming-Hsuan's work [4], he defines 30 types of UI components and models them using the composite pattern to represent their composite structure.

On the other hand, the decorator pattern is also applied to extend the functionalities of UI components. Each decorator represents its additional behavior. The main focus of this chapter will be on utilizing decorators to describe the dynamic behavior of UIs during runtime.

doi:10.6342/NTU202303645



3.1 Categorize Angular APIs

In order to enhance the capabilities of the UI description language to create more modern user interfaces, we decided to draw inspiration from the tools provided by modern frontend frameworks. Then, we analyze the commonly used functionalities by users and abstracted these functionalities as decorators for UI components. This approach allows us to extend the functionalities of our UI components.

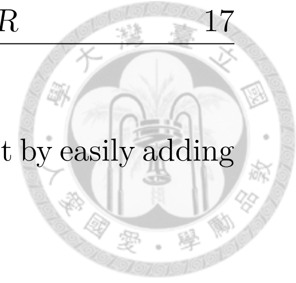
We chose the frontend framework Angular for its provided API. The Angular API [2] enables developers to easily implement certain functionalities of web applications without starting from scratch.

The following are the three categories of Angular APIs we implemented:

Transform displayed value. When the user interface is displaying received data, it often doesn't directly match the desired displayed format. In such cases, programmers are required to write the corresponding code to transform it. Angular provides two types of APIs, called "Pipes" and "Formats," that allow users to easily convert information to the required format for display. For example, these APIs can handle formats such as date and time, capitalization, currency, and more. The commonly used transformations can usually fulfill users' requirements.

Change behavior and appearance based on certain conditions. In most UIs, pages are not in a static state. The page adjusts dynamically over time, altering the elements on the screen. Angular offers two types of directives for implementing these dynamic effects:

doi:10.6342/NTU202303645



- Structural directives: These directives change the DOM layout by easily adding and removing DOM elements.
  - NgIf: Adds or removes elements conditionally.
  - NgSwitch: Renders elements based on the assigned variable.
  - NgFor: Renders a list of elements based on each item in the list.
- Attribute directives: These directives change the appearance or behavior of DOM elements.
  - NgStyle: Updates styles for the HTML element conditionally.
  - NgClass: Adds or removes CSS classes of an element conditionally.

Change certain variable based on user's action. In this part, we will introduce Angular's Validators.

A Validator is a function used to handle forms or a set of input fields. It determines whether the input meets the conditions and returns a validation success or failure result. In Angular, several common Validators are available for users. These Validators include checking whether the input is empty, verifying if the amount of input's characters exceeds (or is less than) certain values, and determining whether the input fits the desired formats. These Validators effectively ensure the correctness of user input before submission, enhancing user experience and preventing erroneous inputs.

doi:10.6342/NTU202303645

3.2 Define Variables in UI Description Language

To enable dynamic changes in the user interface based on conditions, we have identified common functionalities in the previous section. However, expressing these functionalities using our UI description language is challenging. The main difficulty is "how to describe the conditions for UI component changes in a machine-readable manner," rather than relying on natural language descriptions.

Therefore, we analyze the variables within our system. Currently, our description language defines two types of variables: Passing Parameter (the parameters from other pages) and Service Return (the return value of a service). However, these two types of variables might not be sufficient to cover all the conditions required for creating the frontend. UI changes can depend on other conditions, such as displaying an "Input Format Error" message when the entered data is not in the expected format.

Listing all possible conditions is impractical. Therefore, we need to design an extensible variable system within the description language to address potential conditions that might arise in the future.

We've decided to introduce a new variable mechanism within the description language, separating the "source" and "dependents" of system conditions. We will use decorators to bind this variable to UI components, indicating that a change in that component will cause the variable to change. This is referred to as the variable's "source." On the other hand, we will use decorators on UI components to specify that they are "dependents" of variables, meaning that the components' behavior

doi:10.6342/NTU202303645

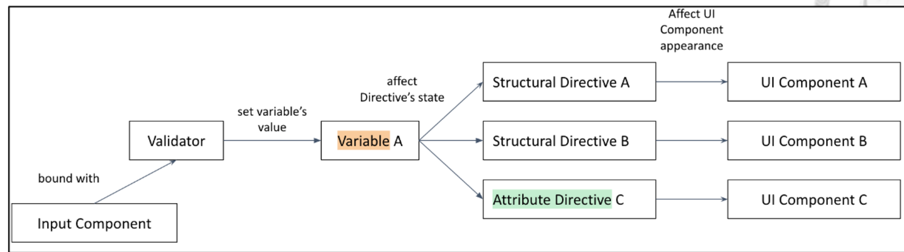


Figure 3.1: Idea of Variable

may change depending on the type of decorators.

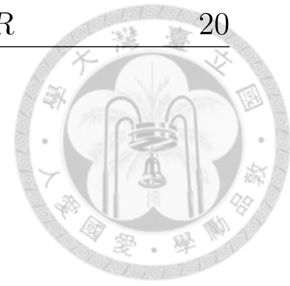
From the example mentioned earlier, this involves two distinct components: an Input component for user input and a Text component that shows an error message conditionally. The Input component serves as the source for this variable, while the Text component becomes the dependent. In Figure 3.1, we can observe that a single variable can have multiple dependents.

3.3 Design and Implementation

Based on the design principles and analysis from the previous two sections, here are the decorators we have defined: (The green part in Figure 3.2)

- Source of the variable:
  - Validator Decorator: Records the conditions that the input component must satisfy and the variable it affects.
  - Service Return Bind Variable Decorator: This decorator is applied to the components responsible for calling services. It declares that specific

doi:10.6342/NTU202303645



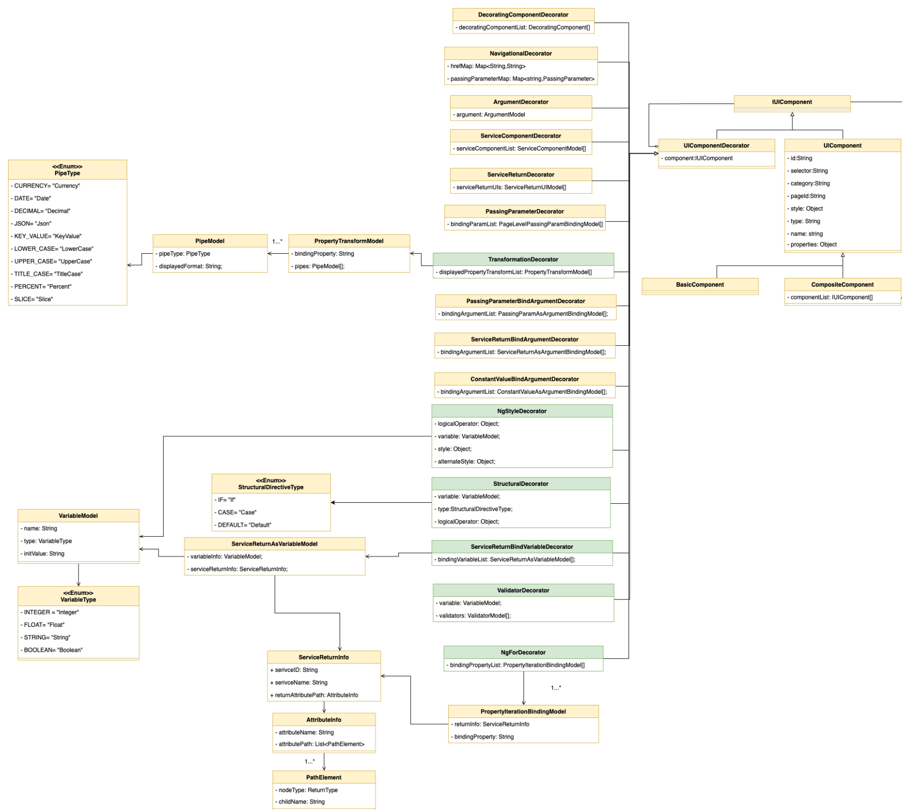
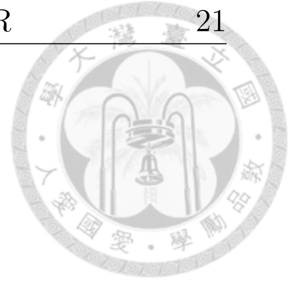


Figure 3.2: Overall UI Component Decorator

doi:10.6342/NTU202303645



return values from the service will be used as variables.

- Dependent of the variable:
  - Structural Decorator: This component displays or hides based on the variable's conditions. We've drawn inspiration from the Angular APIs ngIf and ngSwitch. The logical operation format for variables utilizes Logical JSON operators.
  - NgStyle Decorator: Dynamically sets the element's CSS style based on the variable's conditions.
- Other:
  - NgFor Decorator: If a service's return value is a list, this decorator renders a number of components equal to the length of the list.

Additionally, we will declare the variables in PDL to provide an overall picture of the variables on this page.

doi:10.6342/NTU202303645

Chapter 4

Modeling Tasks

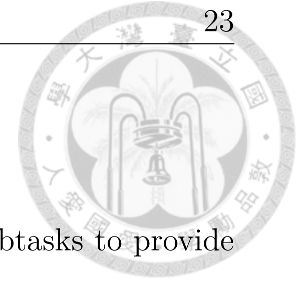
The main focus of this chapter is to integrate the task model into our system. We have decided to use ConcurTaskTrees (CTT) as our task model. First, we need to define and explain each task type and point out their objectives. Lastly, we will model CTT to make it usable within our system.

4.1 Task Type Defining

CTT categorizes tasks into four categories: System, Interaction, User, and Abstraction. Each of these categories can be further divided into more specific types, and each node on the task tree belongs to one of these types.

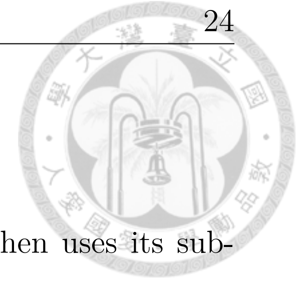
The following are introductions to the defined types. We believe that by applying a combination of these task types, we can effectively express all scenarios a user might encounter when interacting with an UI:

doi:10.6342/NTU202303645



- Abstraction tasks
  - **Task Group**: Summarizes a task and then uses its subtasks to provide detailed descriptions. Note that subtasks do not belong to the same category.
  - **Include Task**: Similarly, it summarizes a task but references other existing CTTs through links. This facilitates "reusability" in our design.
- System tasks
  - **System Task Group**: Summarizes a task and then uses its subtasks to provide detailed descriptions. Note that all of its subtasks are system tasks.
  - **Checking**: The system confirms whether the previous tasks were performed correctly by the user.
  - **Error Message**: The system notifies the user when an error occurs.
  - **Feedback**: The system informs the user about the progress of a task.
  - **Filtering Information**: The system filters data for further operations.
  - **Input Validation**: The system validates the value entered by the user.
  - **Visualize Fixed Value**: The system displays information on the page that remains static.
  - **Visualize Dynamically Acquired Value**: The system displays information on the page that's obtained dynamically.
  - **Service**: The system uses backend services.

doi:10.6342/NTU202303645



- Interaction task
  - **Interaction Task Group**: Summarizes a task and then uses its subtasks to provide detailed descriptions. Note that all of its subtasks are interaction tasks.
  - **Input**: The user enters data into the system.
  - **Select From Fixed List**: The user chooses from predefined options provided by the system.
  - **Select From Dynamically Acquired List**: The user selects options provided by the system, where the options are dynamically obtained.
  - **Select From Visualized Info**: The user chooses an item from information already displayed by the system for further actions.
  - **Control**: The user triggers activities in the system, e.g., clicking, pressing buttons.
  - **Responding Alert**: Appears after alert tasks like Checking and Error Message, indicating the user's response to system notifications.
- User tasks
  - **User Task Group**: Summarizes a task and then uses its subtasks to provide detailed descriptions. Note that all of its subtasks are user tasks.
  - **Problem Solving**: Describes tasks where the user solve issues outside the system.

doi:10.6342/NTU202303645

- **Comparing**: Describes tasks where the user needs to make decision at that moment.
- **Planning**: Describes that the user is planning or making a decision at that time.

4.2 ConcurTaskTrees Modeling

In the previous section, we defined all the task types. In this chapter, we will further model our CTT, as shown in Figure 4.1. We explain the design in this section.

Composite Pattern for Task Trees Firstly, CTT is a tree structure where all parent nodes (parent tasks) encapsulate the descriptions of all their child nodes (subtasks). We use composite pattern to model CTT to express its part-whole hierarchical structure, while treat all objects in the composite structure uniformly.

Association Class for Temporal Operator Additionally, in CTT, there are temporal operators between two tasks. We use association class to represent the relationships between these tasks.

Authentication Further, we apply authentication to our task model to design a system with different roles. In CTT, we need to record two pieces of information.

Firstly, it's necessary to record all actors and the hierarchical structure among them. Different actors have different permissions for the tasks they are allowed to

doi:10.6342/NTU202303645

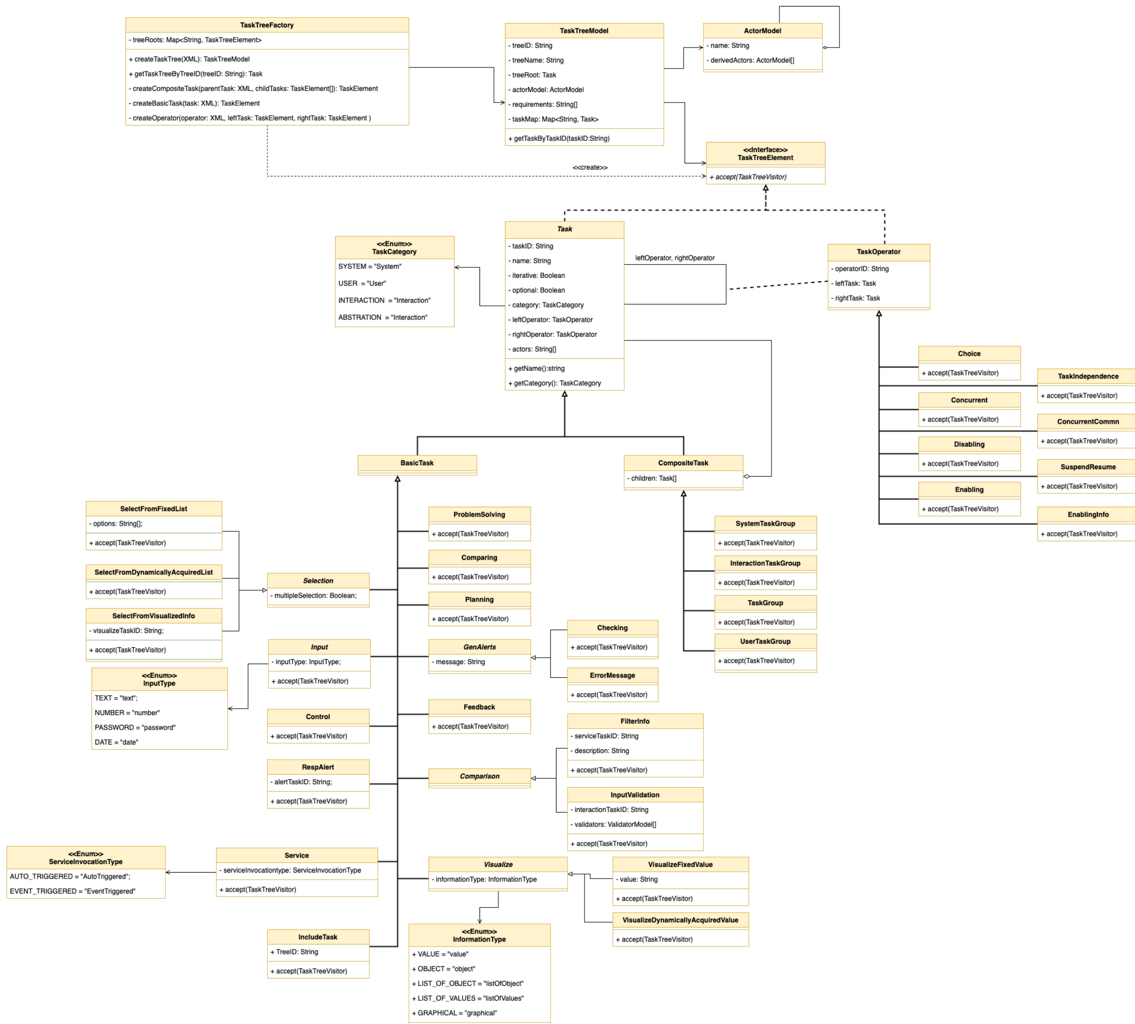


Figure 4.1: CTT Modeling

doi:10.6342/NTU202303645

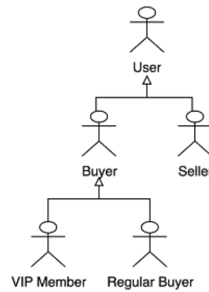


Figure 4.2: Actor's Hierarchical Structure

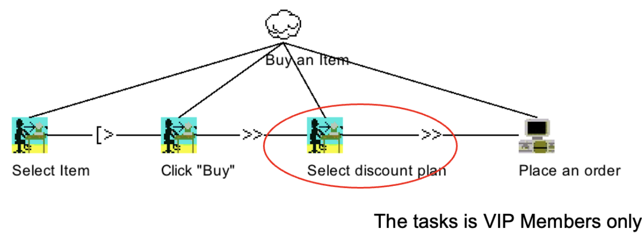


Figure 4.3: Specify Actors within Tasks

perform. These permissions and tasks will depend on the actor's roles and identity, and the information should be appropriately represented within the CTT. (Figure 4.2) is an example of the actor set for an online shopping site.

Secondly, we need to explicitly specify which actors have the permission to perform specific tasks, ensuring the security of the system and proper resource control.(Figure 4.3) is an example for the online shopping site.

doi:10.6342/NTU202303645

Chapter 5

UIDL Generating Algorithm

In this chapter, we will discuss the process of generating our UI description language and utilizing the UI Composition engine improved by Hsu to generate the corresponding code. Figure 5.1 shows the system architecture of the UI generating system.

This chapter will be divided into two parts. Firstly, we will present a process that allows users to transform Use Case Diagrams and Use Case Specifications into

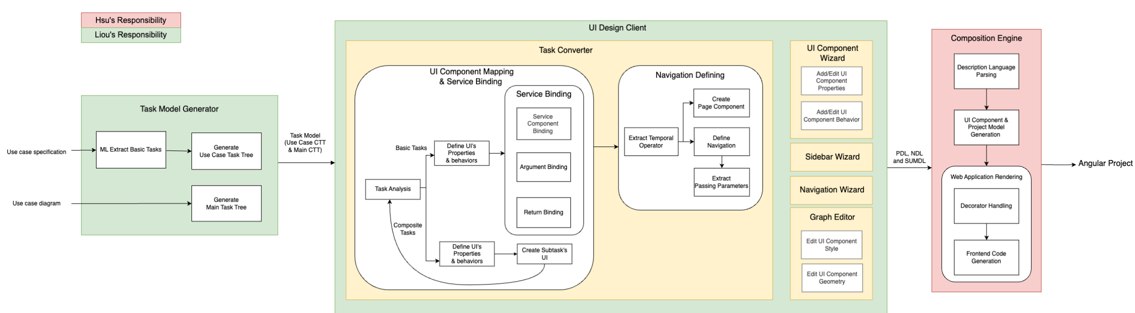


Figure 5.1: UI Generating System Architecture

doi:10.6342/NTU202303645

corresponding CTTs. Since these documents are written in natural language, executing these algorithms, while intuitive, cannot be fully automated. In this section, we will identify the parts where natural language processing and machine learning tools are required to transform the content of the documents.

Next, we will introduce an algorithm to automatically transform CTT into PDL, NDL, and SUMDL. In other words, we will design algorithms to extract UI component information and their behaviors from CTT.

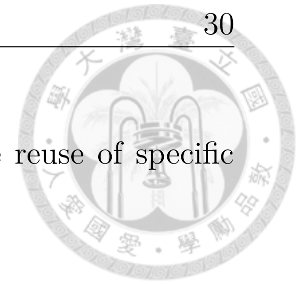
After generating PDL, NDL, and SUMDL, we can utilize the other parts of the UI Design Client to fine-tune our project. We can further adjust the UI style using the Graph Editor, employ the UI Component Wizard to add various components, and adjust properties as well as behaviors of individual components. Additionally, we can use the Navigation Wizard to make adjustments to the UI's navigation.

5.1 Use Case to Task Model

Before discussing the algorithms, we will categorize our self-defined CTT based on usage and discuss what kind of information can be generated from ”Use Case Diagrams” and ”Use Case Specifications,” respectively.

- Main CTT: Main CTT is the task model we've defined to describe the user interface. It may include to other CTTs for use.
- Included CTT: Included CTT represents task models that have been defined and can be referenced. These referenced CTT cannot independently form

doi:10.6342/NTU202303645



a complete user interface but are encapsulated to facilitate reuse of specific functionalities.

We believe that the relationship between a use case diagram and each individual use case is similar to the relationship between our Main CTT and Included CTTs. A single use case cannot form a complete user interface on its own but needs to combine multiple use cases through a use case diagram to create a complete application.

Therefore, our process will begin by transforming the use case diagram into the Main CTT. Then, based on each use case specification, we will create corresponding Included CTTs, also referred to as Use Case CTTs, on a one-to-one basis.

5.1.1 Use Case Diagram to Main CTT

Firstly, we will start by analyzing the information required by Main CTT and what is provided by the use case diagram.

A CTT requires four key pieces of information, including Basic Tasks, Composite Structures, Temporal Operators, and Authentication. For the Main CTT, we only need the information required to link the Use Case CTTs together. Therefore, we will use interaction tasks and IncludeTask type tasks to connect and compose Included CTTs.

Furthermore, composite structures are essential for CTT to enhance readability and avoid the misuse of temporal operators resulting from different precedence. Our approach is to construct the main CTT with a specific pattern, ensuring that the structure of tasks is clear and easily understandable. This approach will benefit us

doi:10.6342/NTU202303645

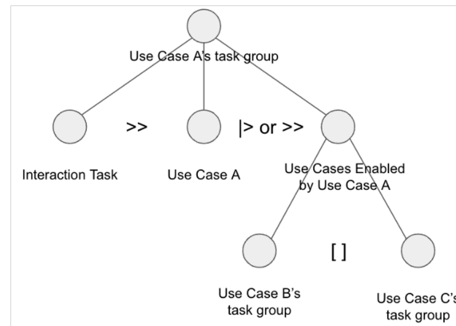


Figure 5.2: Pattern of Main CTT

in interpretation and the generation of UI.

Here is the information that will be provided by use case diagram:

1. Actors: This provides the authentication information for the Main CTT.

2. <<extend>>relation: This information provides the temporal relationships between use cases, thus offering the temporal operator information within the Main CTT.

3. <<include>>relation: This information will not be used. If UC-001 b UC-002, the responsibilities of using UC-2 lies within the CTT of UC-1, not main CTT.

4. The pre-condition and post-condition of each use case.

We have found that the use case diagram provides the required information to construct the Main CTT. We propose a fixed format for creating the Main CTT to ensure the consistency of the tree structure. For each included use case in the Main CTT, it is divided into three parts, as shown in Figure 5.2. The first part is the

doi:10.6342/NTU202303645

interaction task that triggers the use case, describing the user's action to enter this use case. Next, the second part consists of an IncludeTask type task, used to link to the CTT of the use case. Finally, the third part describes other use cases triggered after the completion of this task.

Hence, we have designed an algorithm to recursively create this tree.

- Input:
    1. A task t_A, where t_A references the CTT created for use case A.
    2. A set of post conditions to describe the current state of the system.
- Output: The use cases' task that are enabled by t_A.

Step1: Collect use cases that satisfy one of the condition. Infer from the use case diagram of which use cases can be executed when use case A is completed.

We collect use cases that satisfy one of the following conditions: first, use cases that extend from use case A, and second, use cases whose pre-conditions are satisfied due to use case A.

Take Figure 5.2 as an example, we obtain t_B and t_C.

Step2: Create the task tree for every use case t_j we collected in step1 For each task t_j of each use case j, we will perform three steps to complete its sub-task tree.

1. Identify the action that triggers use case j and create the corresponding interaction task $interact_j$.

doi:10.6342/NTU202303645

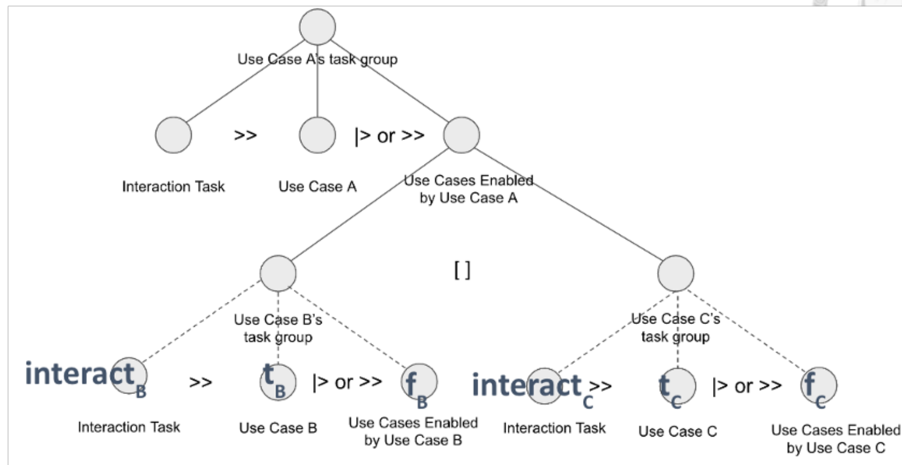


Figure 5.3: Use Cases Enabled by t_A

2. Recursively create the task tree of the use cases that will be triggered when t_j is completed. We'll refer to this tree as f_j.

3. Combine $interact_j$, t_j, and f_j using an abstract task.

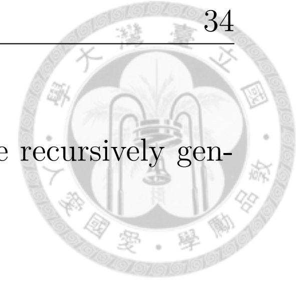
As shown in Figure 5.3, the tasks required for Use Case B and Use Case C are created.

Step3: Connect the task tree created in step2 with "Choice" operator.
Every task group for use case j that we created in step 2 is part of the tasks that can be triggered by t_A. We link them together using the "Choice" operator and then use an abstract task to combine them and return it as the output task tree.

The generated task tree is represented in the subtree labeled "Use Cases Enabled By Use Case A" in Figure 5.3.

To generate the complete Main CTT, we can initiate the algorithm with the

doi:10.6342/NTU202303645



initial state as a post-condition, and the entire Main CTT will be recursively generated.

5.1.2 Use Case Spec to Use Case CTT

For a use case, it often relies on using natural language to describe its implementation details. Therefore, in the process of generating Use Case CTT, we first analyze the information provided in the use case specification and map this information to the four essential pieces of information required to construct a CTT (Basic Task, Composite Structure, Temporal Operators, and Authentication). Then, we pay particular attention to which information needs further processing in natural language and propose corresponding rules, serving as a foundation for training models.

Here is the information provided by our use case specification:

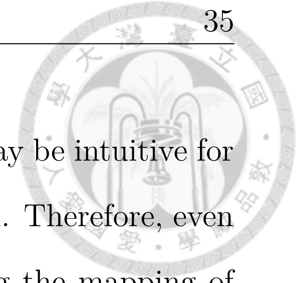
1. List of actions: Our tasks will end when the series of actions is completed, providing information about the basic tasks and the temporal order between them.

2. Related requirements: We store this information in our task model for future use when binding services.

3. Actors of each action: Provides authentication information.

We have identified two main challenges that make automating the derivation of the Use Case CTT from the use case specification difficult:

doi:10.6342/NTU202303645



- The series of actions is written in natural language, which may be intuitive for humans, but the granularity of each action is not guaranteed. Therefore, even though it provides information about our tasks, automating the mapping of actions to tasks is challenging.

- Lack of detailed ”temporal operators” and ”composite structure” information.

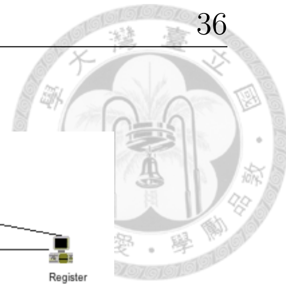
Therefore, we have decided to start by addressing ”how to use basic tasks to describe a user interface.” Once we define the specifications, we can then use natural language processing or let humans map the actions in use cases to basic tasks.

Part1: Define the usage of each task type. We have already provided clear definitions for each task type in the previous chapter and modeled them with the required properties.

Part2: Set restriction to combination of basic tasks. Due to the various combinations of basic tasks, while some combinations might seem reasonable to humans, they might lack certain information for the system. To address this, we have decided to restrict the combination of basic tasks so that they can correctly describe an app.

- Rule 1: There must be an event after the user enters information to enable the system to start performing the preceding system task. In short, an event task is required before the system starts working, as shown in Figure 5.4.

doi:10.6342/NTU202303645



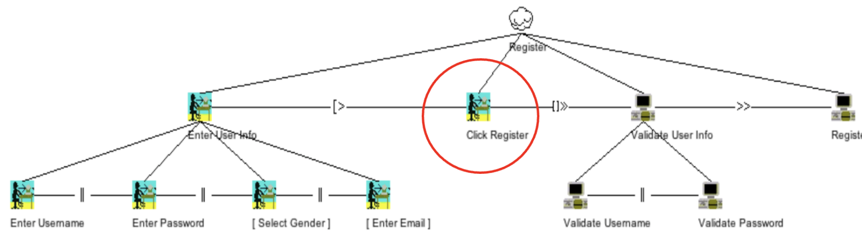


Figure 5.4: Rule 1 for Constructing CTT

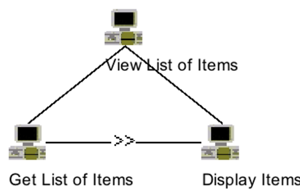


Figure 5.5: Rule 2 for Constructing CTT

- Rule 2: Always get the information before displaying it (regardless of how long ago), as shown in Figure 5.5.

Part3: The tasks should specify their temporal order. Each task has its own ID, and we have decided to use this ID to record the temporal information. The format of the ID is as follows:

{task num}.{branch num}{branch digits}-{concurrent num}

We analyze from right to left:

- For tasks that have the same ”task num,” ”branch num,” and ”branch digits” (if they exist), but different ”concurrent num,” set the ”concurrent” operator between them and group them as a composite task.

doi:10.6342/NTU202303645

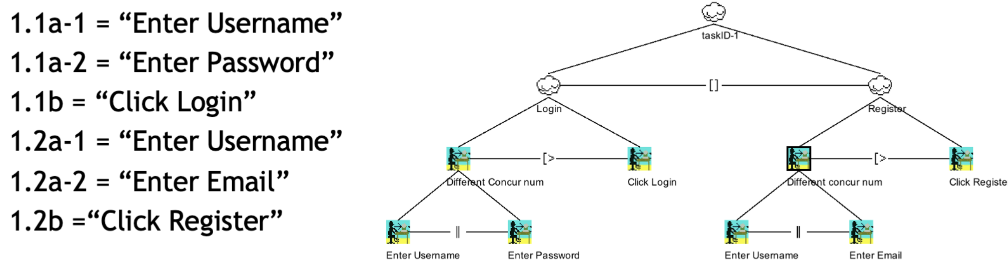


Figure 5.6: Example of Constructing CTT by ID

- For tasks that have the same ”task num” and ”branch num” but different ”branch digits,” set a ”disabling” or ”enabling” operator between them, depending on whether the right task is an event task, and group them as a composite task.

- For tasks that has the same ”task num” but different ”branch num”, set ”choice” operators between them and group them as a composite task.

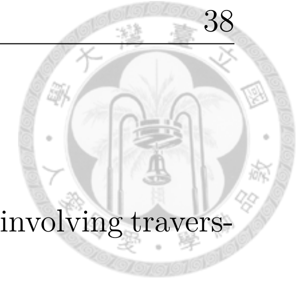
- Finally, for tasks that have different ”branch digits,” set a ”disabling” or ”enabling” operator between them, depending on whether the right task is an event task, and group them as a composite task.

Figure 5.3 is an example where we can observe the results generated based on the task names on the left side of the diagram and their corresponding IDs.

5.2 CTT to UIDL Mapping

In this section, we will discuss how to convert our CTT into PDL, NDL, and SUMDL. The purpose of this transformation process is to extract information about

doi:10.6342/NTU202303645



UI components, navigation, and services from the CTT.

To achieve the goals, we divide the process into two stages, each involving traversing our CTT once:

- Stage 1: Creating UI components and extract their behavioral information.
- Stage 2: Defining navigation between user pages.

At last, we will present the design of our system and display the results.

5.2.1 Retrieve UI Component and Binding Information

This is our first stage of transforming CTT. Table 5.1 is our list of tasks, along with their corresponding UI components and the additional information it provided.

Next, we will traverse our Main CTT using Depth-First Search (DFS). This approach gradually maps the tasks into corresponding UI components and related information using the rules mentioned in the table. Finally, we will generate an intermediate UI to proceed to the next stage.

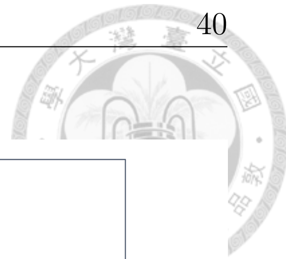
The following example is provided for better understanding. Figures from Figure 5.7 to Figure 5.11 illustrate the process of creating user interface components in the first stage. Figure 5.12 represents the outcome of this stage, where we can observe that navigation and page information has not been included yet.

doi:10.6342/NTU202303645

Table 5.1: Information in Each Task Type

Type	UI Components	Decorator Information	Other Information
Select From Visualized Info			Navigation information
Select From Fixed List	Dropdown, Listbox, RadioButton, Checkbox	Argument	Provide argument information for querying services
Select From Dynamically Acquired List		ServiceReturn, PassingParameter, Argument	
Input	Input, DateTimePicker, Slider, SlideToggle	Argument	
Control	Button, Icon		Navigation information
Responding Alert			
Task Group	Form, Tabs, Dialog, ExpansionPanel		
Visualize Fixed Value	Text , Tree, Table, Card, Image		
Visualize Dynamically Acquired Value		ServiceReturn,PassingParameter	
Checking	Alert, Dialog		
Error Message			
Feedback	ProgressBar, ProgressSpinner	ServiceReturn	
Service		ServiceComponent	
Filtering Information			Filtering data
Input Validation		Add Validator decorator to input control UI component.	

doi:10.6342/NTU202303645



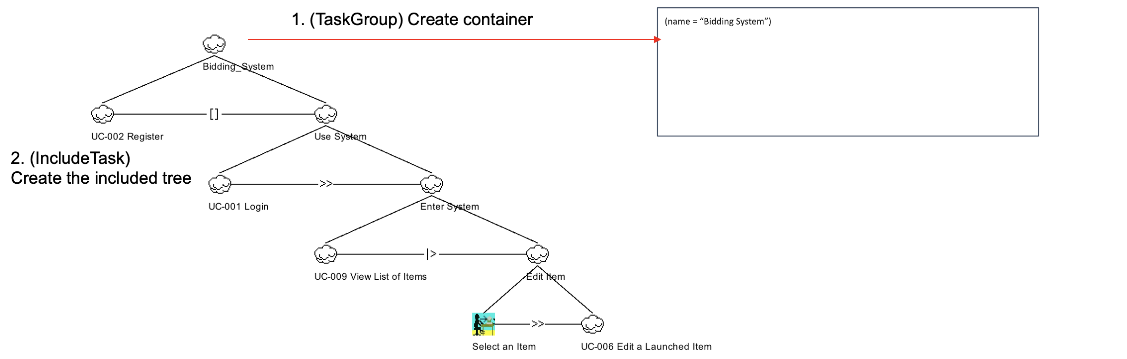


Figure 5.7: Example of CTT to UI Component Mapping(1): Main Tree

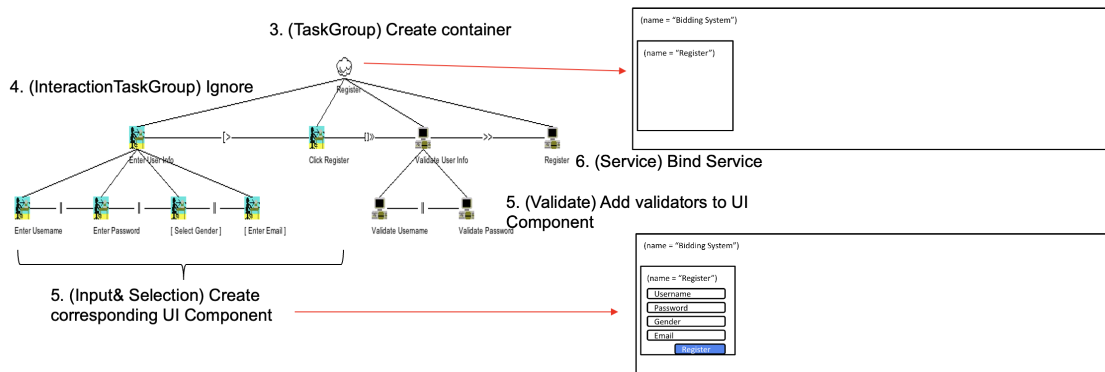


Figure 5.8: Example of CTT to UI Component Mapping(2): Register

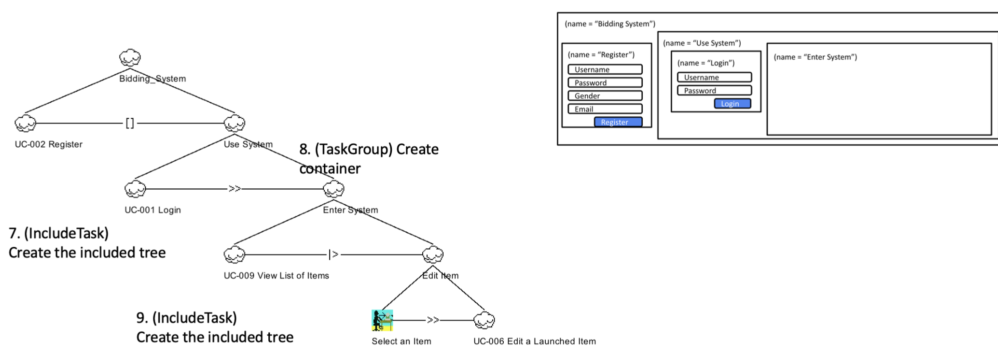
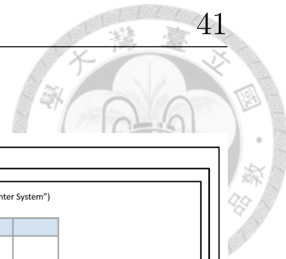


Figure 5.9: Example of CTT to UI Component Mapping(3): Back to Main Tree

doi:10.6342/NTU202303645



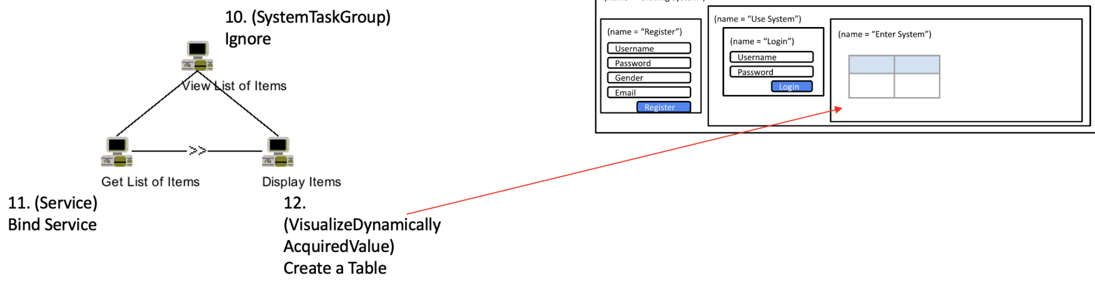


Figure 5.10: Example of CTT to UI Component Mapping(4): View List of Item

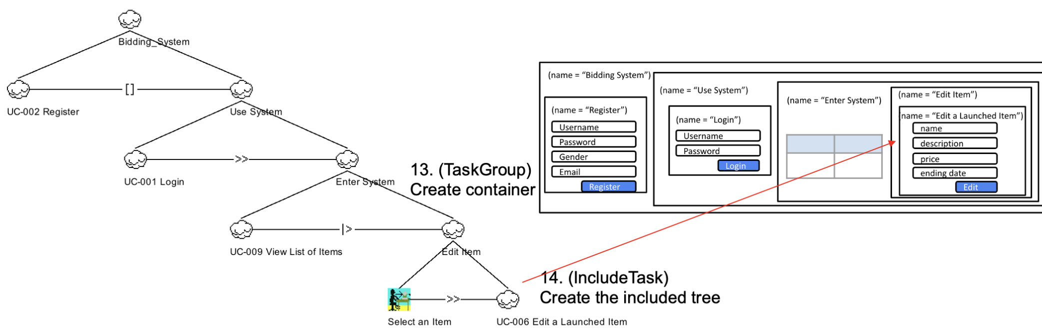


Figure 5.11: Example of CTT to UI Component Mapping(5): Back to Main Tree Again

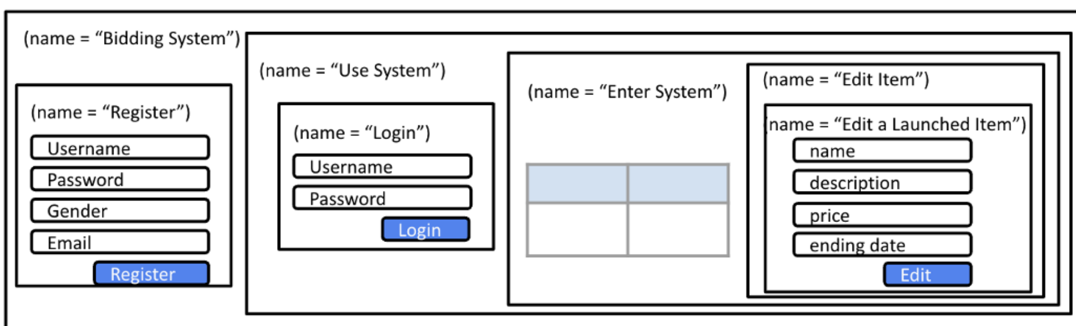
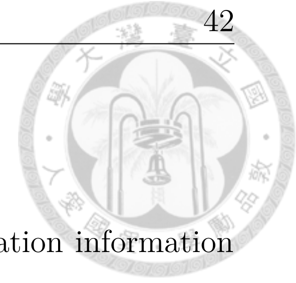


Figure 5.12: Example of CTT to UI Component Mapping: Final Product

doi:10.6342/NTU202303645



5.2.2 Determine Navigation of the UI

In this stage, we will traverse our CTT again to collect navigation information and integrate it into the intermediate product generated in the previous stage. This process involves defining pages and specifying the navigation between those pages.

CTT defines temporal operators between tasks, by interpreting these temporal operators, we can extract navigation information of the UI. Therefore, we start by analyzing the pages and navigation information provided by each temporal operator (refer to Table5.2).

The temporal operators present another challenge that needs to be addressed, which is the precedence between them. The meaning might be different if they are misused. Therefore, when interpreting temporal operators, it's important to pay special attention to their precedence. Furthermore, we observe that the tree-like structure of the CTT can be considered as ”parentheses” in mathematical expressions, which is helpful in determining the combinations and hierarchical relationships between tasks.

We use the following concept to interpret the navigation information in CTT:

- During the traversal process, when encountering operators, decide whether to create a page for the previously traversed UI components.
- After the UI components on the right side of the operator are generated, define the navigation between the two user interfaces. For example: 1 + 2 * 3 (the + operation will be performed only after the operation of 2 * 3 is completed).

Since we want to traverse CTT to ”calculate” a project, and the CTT’s operator

doi:10.6342/NTU202303645

Table 5.2: Information in Temporal Operators

<table>
<tr><th>Name</th><th>Temporal Information</th><th>Page Information</th></tr>
<tr><td>Choice</td><td>The generated sub UI can be connected through “tab” or “sidebar” component.</td><td rowspan="4">None</td></tr>
<tr><td>Task Independence</td><td>Separate the generated UI components using tabs or create a dedicated page responsible for navigating to each UI. After the task is completed, it should navigate back to the previous page.</td></tr>
<tr><td>Concurrent</td><td rowspan="2">This implies that the generated subUI of tasks must be in the same page.</td></tr>
<tr><td>Concurrent Communicating</td></tr>
<tr><td>Disabling</td><td>The subUI that is generated by the task ”after the second task” is not on the same page.</td><td rowspan="2">Create a page that includes the right task’s UI and the UI that is created before.</td></tr>
<tr><td>Suspend-Resume</td><td>The subUI that is generated by the task ”after the second task” is not on the same page. Once the target task is complete, the UI should navigate back to the source task’s UI.</td></tr>
<tr><td>Enabling</td><td rowspan="2">If both tasks can generate UI, the second task’s UI appears after the first task’s UI.</td><td rowspan="2">Create a page that includes the left task’s UI and the UI that is created before.</td></tr>
<tr><td>Enabling with information passing</td></tr>
</table>

doi:10.6342/NTU202303645

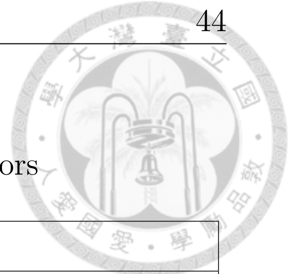


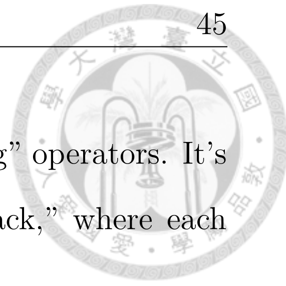
Table 5.3: Behaviors when Pushing/Popping Operators

Name	Create Page(when pushing)	Behaviors when popping operators
Choice	No	Pop two elements from the operand stack, combine them using a Sidebar, and then push the resulting ”Continuous Page” into the stack.
Task Independence	No	Pop two elements from the operand stack, combine them using a Tab component, and then push the resulting ”Continuous Page” into the stack.
Concurrent	No	Pop two elements from the operand stack, place them into a list, and then push the list into the stack.
Concurrent Communicating	No	(Same as above)
Disabling	Yes, include the event task's UI component.	Pop two elements from the operand stack, navigate the first element to the second element, and then push the resulting ”Continuation Page” into the stack.
Suspend-Resume	Yes, include the event task's UI component.	Pop two elements from the operand stack, navigate the first element to the second element, navigate the second element back to the first element, and finally push the resulting ”Continuation Page” into the stack.
Enabling	Yes	Pop two elements from the operand stack, navigate the first element to the second element, and then push the resulting ”Continuation Page” into the stack.
Enabling with information passing	Yes	(Same as above)

is similar to mathematical expression, we will apply Dijkstra's Two-Stack Algorithm with some additional step on task tree.

In this algorithm, we decide whether to ”create a page” and establish the ”nav-

doi:10.6342/NTU202303645



igation between two user interfaces" when "pushing" and "popping" operators. It's important to note that the operand stack is the "component stack," where each stack element holds a list of UI components.

Thus, in our system, we have two stacks: the "temporal operator stack" and the "UI Component Stack." The elements in the "UI Component Stack" not only record a list of UI components but also contain information about components that can perform navigation.

Here, we introduce a new term called "consecutive page," which refers to a series of pages that have defined navigation between them. The largest consecutive page is the complete UI, which is what we aim to achieve through the traversal of the CTT.

Figures from Figure 5.13 to Figure 5.21 illustrate the process of applying the second stage algorithm to the UI components created in the first stage. The resulting UI shown in Figure 5.22 is generated based on the information from the main CTT.

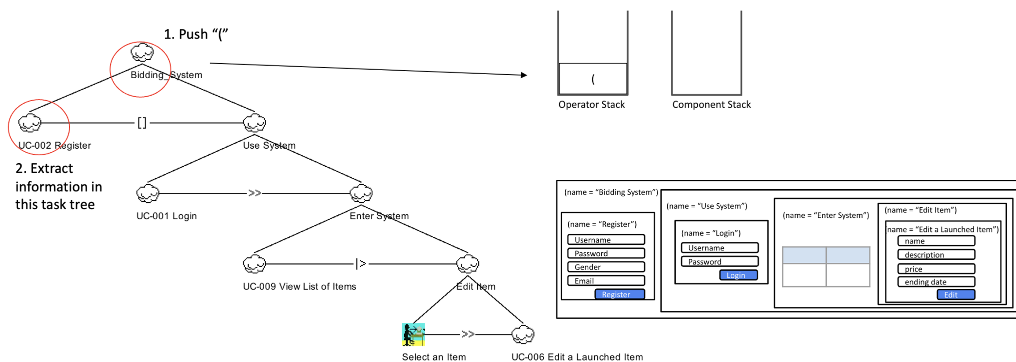
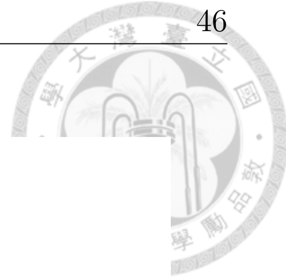


Figure 5.13: Example of Determine Navigation(1): Main Tree(1)

doi:10.6342/NTU202303645



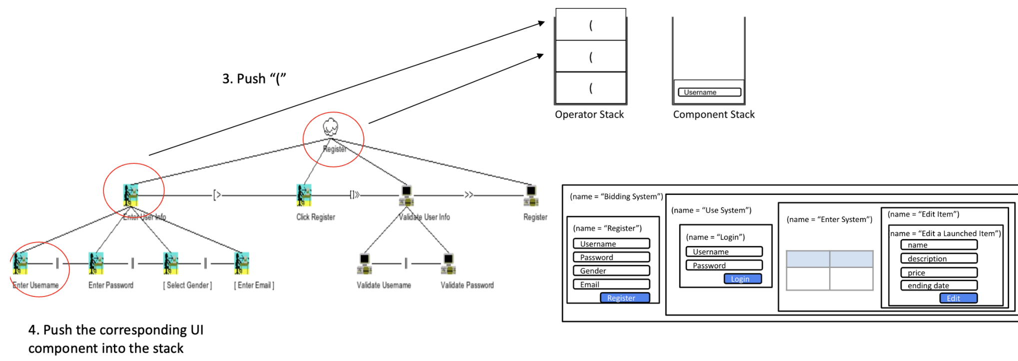


Figure 5.14: Example of Determine Navigation(2): Register

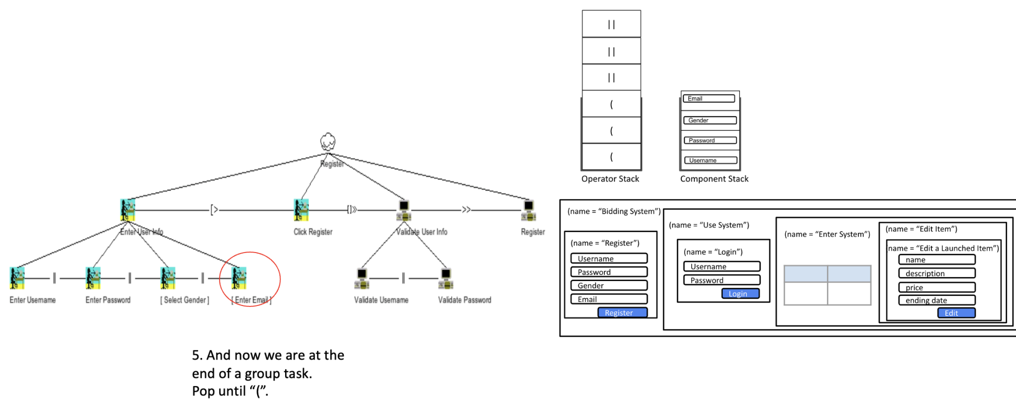


Figure 5.15: Example of Determine Navigation(3): Register

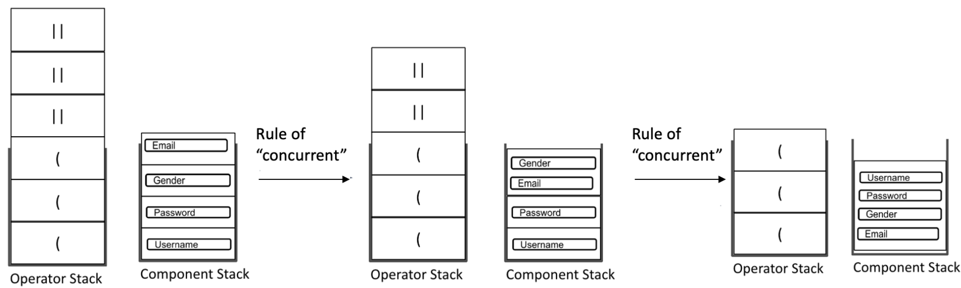


Figure 5.16: Example of Determine Navigation(4): Register(Pop Operator Stack)

doi:10.6342/NTU202303645

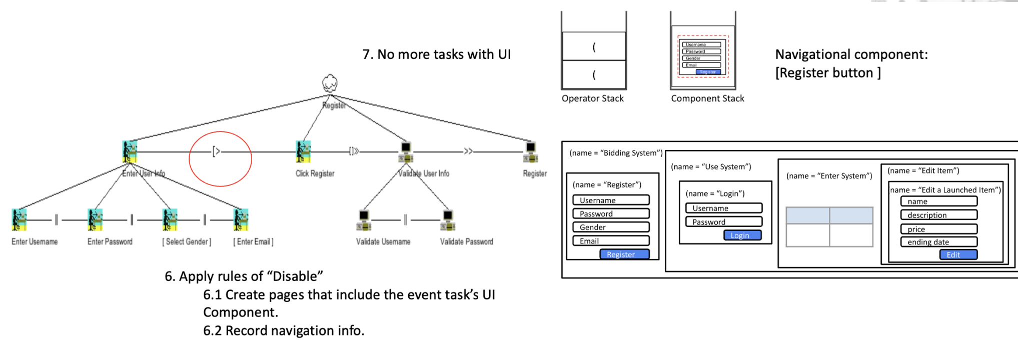


Figure 5.17: Example of Determine Navigation(5): Register

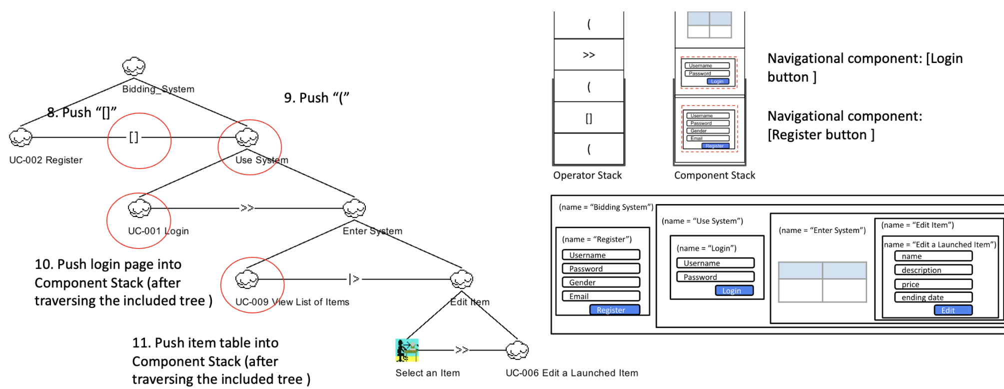


Figure 5.18: Example of Determine Navigation(6): Main Tree

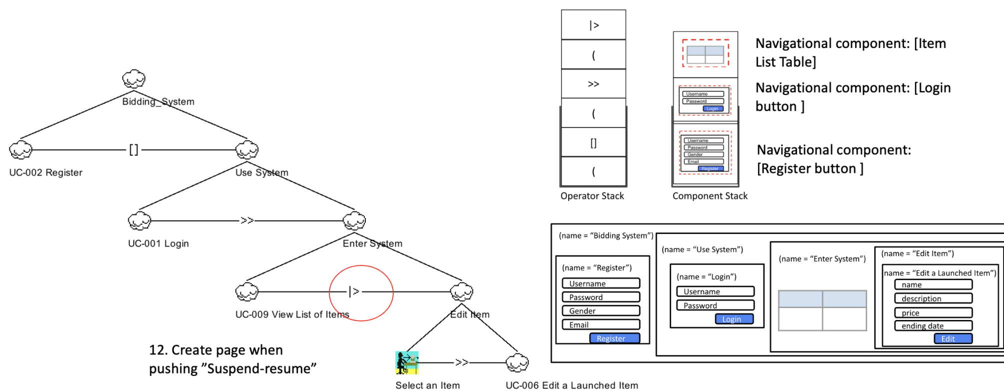


Figure 5.19: Example of Determine Navigation(7): Main Tree

doi:10.6342/NTU202303645

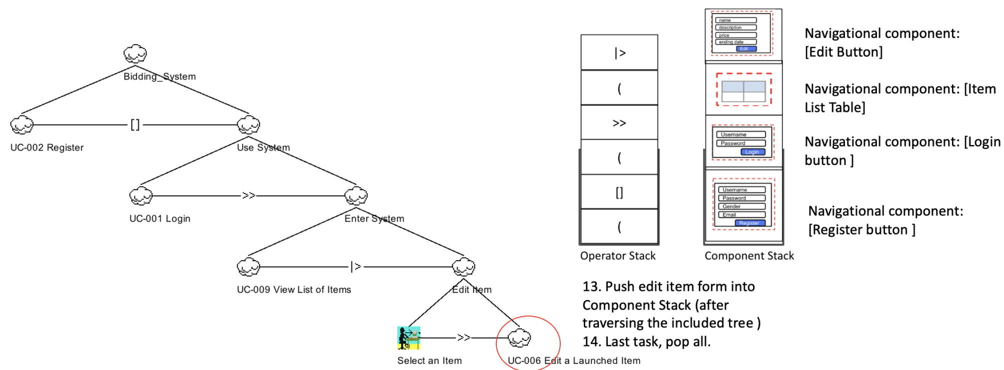


Figure 5.20: Example of Determine Navigation(8): Main Tree

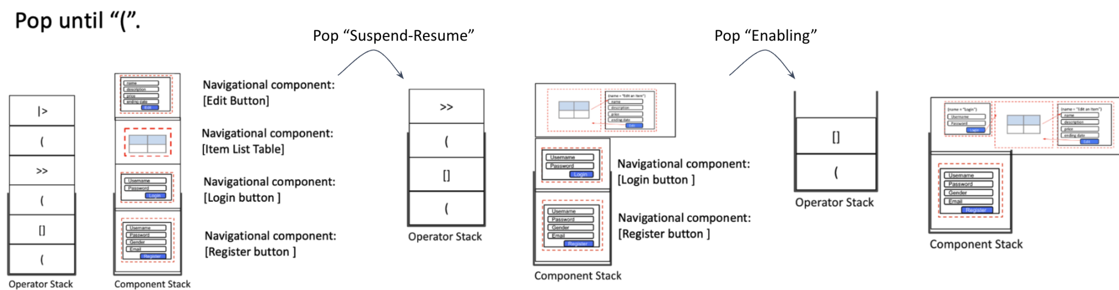


Figure 5.21: Example of Determine Navigation(9): Main Tree(Pop Operator Stack)

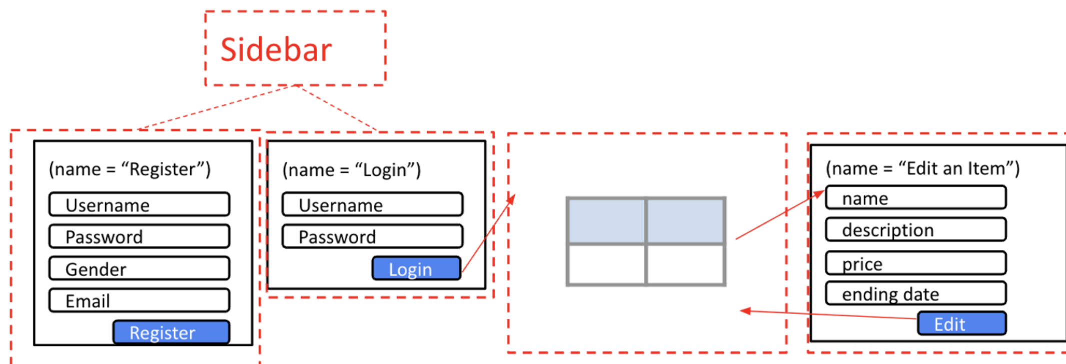


Figure 5.22: Example of Determine Navigation: Final Product

doi:10.6342/NTU202303645

5.2.3 Design on Task Converter

Finally, we implement the transformation algorithm that converts CTT into PDL, NDL, and SUMDL. We explain the design in this section.

We apply two design patterns:

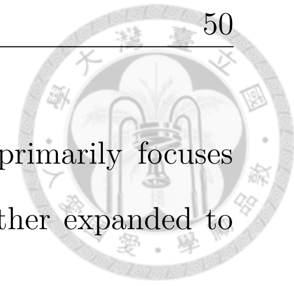
- **Visitor pattern**: In our transformation process, we traverse the CTT twice to gather UI component information and navigation information. Since we collect different information from the CTT depending on different tasks and operations, we apply the visitor pattern to traverse the CTT object structure (see Figure 5.23).

  Here we have proposed three visitors.

  - **Task Tree Validation Visitor**: This visitor is responsible for validating

Figure 5.23: Visitor Pattern on CTT

doi:10.6342/NTU202303645



the CTTs to ensure their correctness. Currently, it primarily focuses on verifying the attributes of tasks, and it can be further expanded to validate more parts of the CTT in the future.

- **UI Component Mapping Visitor**: This visitor is responsible for the first stage transformation described in this chapter. As it traverses the CTT, it maps the tasks into UI component builders along with the necessary information for the builders. It's important to note that this visitor is also responsible for binding services to UI components.

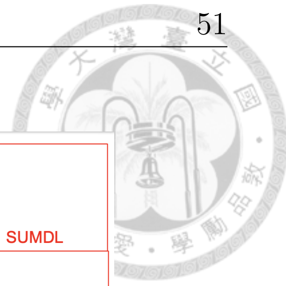
- **Navigational Visitor**: This visitor is responsible for the second stage transformation described in this chapter. It creates page related builders and defines navigation as it traverses the CTT.

• **Builder Pattern**: Different properties and behaviors (defined using the description language) are added to different UI components in different stages. To gradually add the information obtained by visitors into UI components, we decided to use the Builder Pattern to create various UI components. This allows us to obtain UI data at different times during the traversal by the two visitors. At the end of the traversal, the required UI components and description language are generated(Figure 5.24).

We have three types of builders:

- **UI Component Builder**: This builder is responsible for receiving data about UI components and we have different builders for various types of UI components. The Page Builder, in particular, is responsible for

doi:10.6342/NTU202303645



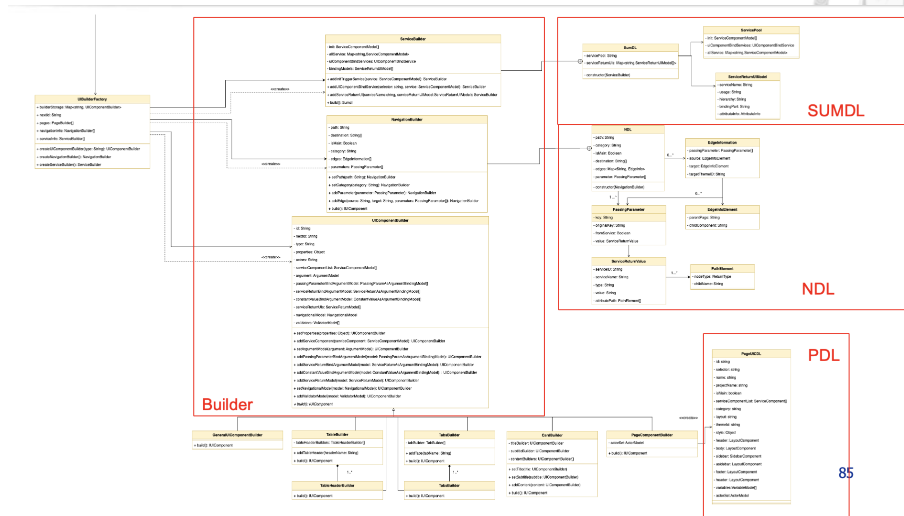


Figure 5.24: Builder Pattern for UIDL

generating the PDL.

– **Service Builder**: This builder is responsible for handling data related to services, which includes the utilization of service components and their return values.

– **Navigational Builder**: This builder is responsible for receiving navigational information.

doi:10.6342/NTU202303645

Figure 5.25: Task Converter

doi:10.6342/NTU202303645

Chapter 6

Conclusion

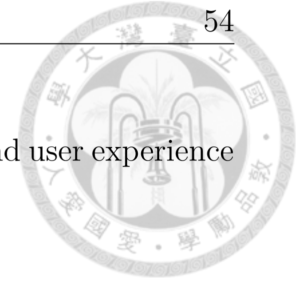
6.1 Summary

We have successfully proposed a method to save time and improve intuitiveness in frontend development by incorporating the Task Model into our UI generation process. This involves defining the task model, proposing a method for creating the task model, and completing the transition from the task model to the UI.

Even though the process of transforming use cases into a task model cannot be fully mechanical (due to use cases being composed in natural language), we have devised an intuitive approach to generate the task model from the use case.

Furthermore, by following the restrictions on constructing the task tree, the task tree may not necessarily be created from the use case. This means that we have greater flexibility in generating task models, allowing us to better align with requirements and user expectations. These methods and techniques make our development

doi:10.6342/NTU202303645



process more efficient and flexible, thereby enhancing the quality and user experience of the user interface.

6.2 Future work

As for future work, we will discuss two topics:

Generating CTT In the step of generating CTT, it's not fully automated at this point. Training natural language processing models is necessary to enable them to convert actions from use cases into models of basic tasks.

UI Composition Currently, in the UI Composition process, we parse the necessary data from PDL, NDL, and SUMDL, and then utilize predefined templates to generate the corresponding code. In the future, our goal is to develop a process that can transform these three UIDLs into corresponding abstract syntax trees (ASTs), thereby creating a more generalized representation. This approach would contribute to a more versatile and adaptable representation of user interfaces.

doi:10.6342/NTU202303645

Bibliography

[1] M. Abrams, C. Phanouriou, A. L. Batongbacal, S. M. Williams, and J. E. Shuster. Uiml: an appliance-independent xml user interface language. *Computer networks*, 31(11-16):1695–1708, 1999.

[2] Angular api. `https://angular.io/api`.

[3] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with computers*, 15(3):289–308, 2003.

[4] M.-H. Hsieh. Construct and bind user interface components. Master's thesis, National Taiwan University, 2021.

[5] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. López-Jaquero. Usixml: A language supporting multi-path development of user interfaces. In *Engineering Human Computer Interaction and Interactive Systems: Joint Working Conferences EHCI-DSVIS 2004, Hamburg, Germany, July 11-13, 2004, Revised Selected Papers*, pages 200–220. Springer, 2005.

[6] F. Paterno', C. Santoro, and L. D. Spano. Maria: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 16(4):1–30, 2009.

doi:10.6342/NTU202303645

[7] Usability.gov. https://www.usability.gov/how-to-and-tools/methods/user-interface-elements.html.

doi:10.6342/NTU202303645