Automated Model-based analysis of Web-based User Interface (UI)

Xiaoyu Cui

thesis nr: INF/SCR-08-33

February 11, 2009
9 Case study
  9.1 wiki website UI specifications .......................................................... 51
  9.2 wiki website UI model ........................................................................ 52
  9.3 Verification ....................................................................................... 53
  9.4 Testing .............................................................................................. 54
  9.5 Memory leak detection ....................................................................... 54

10 Conclusion ......................................................................................... 58

11 Appendix
  11.1 Appendix 1 ..................................................................................... 59
  11.2 Appendix 2 ..................................................................................... 61
Chapter 1

Introduction

1.1 Motivation

Many companies have moved to use web applications to run their business (interact with their customers, internal administrations, business scheduling, etc) and to provide global access over Internet to their business.

Web application is very attractive, in particular when compared to traditional client-server applications. A web application is platform independent. Since all its functions can be accessed via a web browser, a web application requires no effort to distribute. Furthermore, releasing a new version only requires changes at the server side. So, installation, upgrade, and maintenance efforts are minimized. It is also supported by many technologies e.g. Javascript, DHTML, Flash, Ajax enabling much better interactive experience than before (e.g. by playing audio, drawing on screen, etc).

Users’ acceptance is an important measure to a system’s success. The functionalities that a system provides to users are obviously very important, but the way that the system delivers its functionalities to user is also important. Users need to interpret the system’s output and provide input to the system. The system should furthermore enable users to be aware of (the relevant part of) its state and control behaviors.

Web-based User interface (Web-based UI) allows a user to interact with a web application. It is usually the only entry point for users to the web application, hence its good design is crucial the web application’s availability. User may simply give up using a web application that is difficult to use, even the web application itself is technically superior and provide excellent functionalities. On the other hand a good UI makes a system easy to understand and to use, hence enabling users to do their tasks more accurately and efficiently and to require less training and less technical support.

Web-based UI design concerns features like how well the output of a web application can be understood by users, how well the manipulation flow affects users’ task progress, how well the web application can prevent users from giving wrong input, and so on.

To some extent it is true that Web-based UI is not as critical as the core business logic of an application. But for employees on the working floor poor interface may significantly impact their productivity and stress level (and there are also cases where incompleteness in the user interface caused major accidents, e.g. the USS Vincennes disaster[19]).

Developing a good Web-based UI is however not trivial. A web-based UI is often more complicated that the logical functionalities behind it. To facilitate human interaction it has to wrap these logical functionalities with complex networks of widgets (forms, menus, buttons, links, tables, etc). Programmers may also pay too much attention on how to make codes more elegant and their algorithms more efficient, focus on specifying back-end requirements, validating specification, testing codes and prove the correctness of implementation but lack of consideration about if the user interface can be well understood by users. It may result that when the whole system gets into testing
phase, it happens that a user comes along and breaks it.

To ensure the user can successfully interact with the UI, the UI engineers are involved in development lifecycle. Those UI engineers normally cannot program, they just focus on the UI design. They can work independently without back-end implementations and visualize UI design in pictures. Even when the web-based UI is well designed and implemented, at test phase, the UI engineers still need to interact with real web application to check if the UI design is implemented correctly. This is normally done by manual testing.

Most web applications provide real-time information like promotion information, new products information, etc. To keep those information up to date, it requires their web-based UI are updated frequently (contents updated, new links and pages added, redesign layout, etc).

A web-based UI consists a set of web pages. Those web pages are related together by e.g. links and buttons. Before creating those web pages, there are UI specifications defined for describing the style, structure and relations between pages like consistency, etc. The size of UI specifications can be very huge, there may have tons of UI specifications which results in 150 pages guideline. So it is very easy to make mistakes in UI design.

There may also be relations between pages where things can easily go wrong; e.g. when a page A that should only become accessible after a user visited a page B. When there is a large amount of web pages to maintain, with more than one engineer to maintain them, and when pages are updated frequently, ensuring that such a rule holds can be very difficult. E.g. the lack of communication between engineers may lead to inconsistency between pages. Furthermore, when a new engineer is assigned the task of working on some web pages, it may require a lot of time for him to figure out what the rules are that he should watch out.

Above all, the methodologies for evaluating the specifications of web-based UI are not efficient. That is, they require lots of time and manual works but are only limitedly successful in exposing inconsistencies (between UI implementations and their specifications):

- The UI engineers have to keep all UI specifications in mind when they design UI. Therefore, it may be very easy to make mistakes when many UI specifications defined.
- After software engineers implement a web application, at test phase, extra manual efforts are still required by UI engineers for validating UI specifications.

With the rapid growth in number and complexity of web applications, we would need a better tool to avoid those mistakes.

### 1.2 Problems

The mistakes mentioned above can be avoided by automating the analysis of UI design and implementation (whether they satisfy the given UI specifications).

In this thesis, we will demonstrate a model-based analysis[15] for web-based UI. In such an analysis, we carry out the analysis based on a model which describes an actual system. Such a model is abstract and in itself requires no real implementation. This allows errors to be detected early in the design phase. Within such an approach, there are several problems we have to solve:

- How to model a web-based UI?
- How to represent UI specifications which can be analyzed based on model?
- How to check the specifications?
- Although models are abstract, models of real UIs are likely to be still complicated; constructing and maintaining them are then a problem. So, how can we make the construction of models easier and more automatic?
• How to integrate our approach into a real development lifecycle?
• How to automate testing UI specifications against real implementations?

1.3 Approach overview

Here is an overview of how our approach works (in order to solve the problems mentioned above):

• Given a web-based UI to develop, we first we construct models. A web-based UI can be viewed as a nondeterministic system: from each web page \( p \) there are multiple next pages possible by following the links in \( p \). We use a Kripke structure[8] (which is a type of nondeterministic finite state machine) to model such a system. In this thesis we use our own extended Kripke structure specifically to represent web-based UIs. Even a model can become quite complicated. So, directly constructing it with e.g. a text editor would be very time consuming, difficult and error prone. To overcome this, we developed a graphical UI builder (GUI builder) that allows an engineer to visually design a UI, and from which a UI model can be automatically generated.

• Then we express our specifications. A logic called Computation Tree Logic (CTL) is used to express UI specifications. It is a (branching-time) temporal logic; it can be used to reason about the behavior of a nondeterministic system over time over time [16].

• We check the specifications. We offer two ways. The first is by using a model checking tool called SMV [4]. Model checking is an approach to automatically verify a finite state system against a given specification [8]. There exists some model checking tools. SMV can check CTL specifications. However, since the system should be described SMV’s own language, we also provide a translator to convert our extended Kripke structure to SMV’s language.

The above way of checking only checks if a given model satisfies our specifications. It does not check against the real implementation. To do the latter we provide a way to generate test cases from a UI model. We then can apply those test cases to a real implementation with a tool called Selenium.

1.4 Contribution

The contributions in this thesis are:

• We propose an extension to Kripke structure which is more suitable to model web-based UIs (will be called UI models).

• We show how to represent UI specifications in CTL.

• We propose a scheme to translate our UI models and its CTL specifications to the input language of the model checker SMV so that we can do verification (on the models).

• We propose a scheme to automatically generate tests from UI models. Unlike the model checking approach (above), the tests are to be exercised on the real UI. We extend an existing tool called selenium[3] to invoke real tests.

• In addition to more traditional examples of properties to validate, we will also demonstrate the flexibility of our approach by using it for detecting memory leaks in browser when running web application inside.

• With our approach, we demonstrate a new way of developing and maintaining web-based UI which is more efficient and effective than the current practice.

• We deliver a prototype implementing all proposals mentioned above.
Chapter 2

Prototype Overview

We constructed a prototype as a proof of principle that our approach works. For explaining the approach itself, rather than starting from the theory, it seems easier to start by explaining how the prototype works. It will give a more concrete view of how our approach works.

The first step of our approach is constructing UI models. A complete UI model includes all UI possible states, the transitions between those states, and how the UI looks like at each state. The last is represented by UI elements and UI attributes.

Constructing a UI model includes two parts. The first part is designing the basic structure of the UI which should consider all possible UI states, the transitions between those states and what the UI will mainly represent at each state. The second part is to give the visual design for each state.

In the prototype, we implement a GUI builder which allows an engineer to create states, specify transitions and visually do their design. With the structure of the UI in hand, the UI engineer proceeds with the GUI builder to visually construct the details of the UI at each state. Below is a screenshot of designing a wiki home page at Information department of Utrecht university.

As shown in above screenshot, there are 9 main feature in GUIBuilder:
• State and Model controls It enables to create a new state, delete state, import a state, import model and export current model to a xml file.

• State selector It enables to switch between different states.

• Create & Delete element It enables to create different elements and delete certain element.

• Element attributes displayer & editor It enables to display the selected element attributes and edit selected element attributes.

• Design area It enables to select elements, reallocate elements and shape elements.

• Output message console It shows the output message like the exported UI model in XML format.

• Transitions editor It enables to edit transitions between states.

• Initial states editor It enables to specify the initial states of the UI model.

• Actions editor It enables to edit the actions includes action type, element locator and action content.

Given some specifications (in CTL), the next step is to verify if the UI model as obtained above satisfies the specifications. The CTL specifications consists of CTL formulas and are encoded as Java classes. The UI model in XML is then turned to a Java model too (a module called UIModel-XMLLoader does the task), which is then filled with some statistic information. The resulting model is called flattened UI model (it has the same structure as the original UI model, but enhanced with statistic information). An SMV converter then takes both UI flattened model and UI specifications to convert them into SMV input language. At the testing phase, we need to integrate UI model and another open source tool together, and that open source tool is implemented in Java. So we implement UI model and UI flattened model generator in Java too. Finally, the SMV model checker takes SMV program and specifications produced by the SMV converter and returns the verification result.

The last step of our approach is testing real web applications against UI specifications. We have a Test Generator that generates test cases from the UI flattened model. Each test case is a sequence of actions. Then the test cases together with UI specifications are passed to the Test Launcher. We use a tool called Selenium to perform the actions (like click links, fill input box, etc) on a real web
The Test Launcher will setup Selenium’s run-time environment and convert the actions defined in test cases into Selenium commands so that Selenium can invoke those actions on a real application.

Selenium can retrieve information from the real application, e.g. which widgets are presented. The Test Launcher can run Selenium tests as JUnit tests. It can assert UI specifications against information returned from Selenium. In the end, a test report can be automatically generated from JUnit tests.

Furthermore, from a UI model the Test Generator can generate an infinite sequence of actions. By doing this, we can stress the web application for a long time to see e.g. its memory usage and possible memory leaks. As an example for such an analysis we use a memory monitoring tool called sIEve. It can run a web application inside it, and monitor the memory usage of that web application. But native Selenium cannot run inside sIEve, so we extend both Selenium and sIEve to bridge them together.
Chapter 3

UI model

3.1 Kripke structure

A Kripke structure[5] is a non-deterministic finite state machine which describes a system’s behaviors. The states of this machine have no internal structure. They abstractly represent the system’s actual states (which have internal structures), and the transitions between the machine’s states abstractly represent actual state transitions in the system. An interpretation function can be used to express properties that hold in various state.

Definition 3.1 A Kripke structure $K$ is represented by a tuple $(S, I, R, L)$ where:

- $S$ is a finite set of states.
- $I$ is a set of initial states such that $I \subseteq S$.
- $R$ is a set of pairs representing the set of transitions of $K$. It is such that if $(s, t) \in R$ then it means there is a transition in $K$ that goes from the state $s$ to the state $t$. a function-like notation is introduced, so that $R(s)$ denotes the set of all ‘next-states’ of $s$. More precisely, $R(s) = \{t \mid (s, t) \in R\}$
- $L$ is an interpretation function. An interpretation function maps each state of $K$ to ‘properties’, encoded as a set of atomic propositions that is considered to hold in that state. For instance, suppose there is a set $\{\text{isRed}, \text{isGreen}, \text{isBlue}\}$ of atomic propositions, then the properties of a certain state $s$ can be $\{\text{isRed}, \text{isGreen}\}$, etc.

A web application can be represented by a Kripke structure. Each web page can be viewed as a state of the Kripke structure. State changes triggered by clicking links or buttons can be viewed as transitions in the Kripke structure. Each web page may contain some interesting properties, like the amount of links, the amount of words, etc. Those properties can be represented in an interpretation function $L$.

Example 3.2 Consider a simple a web application with three pages: home, login, and mailbox. From the home page, there is a link to the login page. The login page links back to the home page and also links to the mailbox page; and the mailbox page links to home page. Further, at each page, the amount of links is considered as the property for that page. The Kripke structure representing this web application can be defined as follow:

- $S = \{\text{home}, \text{login}, \text{mailbox}\}$
- $I = \{\text{home}\}$
- $R = \{(\text{home}, \text{login}), (\text{login}, \text{home}), (\text{mailbox}, \text{home}), (\text{login}, \text{mailbox})\}$
- $L = \{\text{home} \mapsto \{\text{links} = 1\}, \text{login} \mapsto \{\text{links} = 2\}, \text{mailbox} \mapsto \{\text{links} = 1\}\}$
3.2 Computation tree logic

Temporal logic is a formalism for qualitatively describing and reasoning how to assert system properties over time. It extends the propositional logic by the temporal modalities for expressing properties about the relations between the state labels in the executions of system. Temporal logic imposes the temporal modalities on top of the propositional logic. The temporal modalities can be defined as the operators like:

- F: “eventually in the future”
- G: “always in the future”

The time concerned by the temporal logics can be either linear or branching.

- Linear temporal logic states that there is only one possible next state for each current state. The semantics of a linear temporal logic is defined in terms of an infinite sequence of states. Example 3.3 depicts a infinite sequence of states which is constructed based on example 3.2.

- Branching temporal logic states that there are multiple possible next states for transition, this can be thought as bringing in multiple time lines. The semantics of a branching temporal logic is defined in terms of an infinite, directed tree of states called computation tree. Each traversal from the root of the tree represents a path so the computation tree represents all possible paths. The computation tree can be established by unfolding the states by starting is an initial state and then following the transitions described in the Kripke structure. Example 3.4 depicts a computation tree which is established based on example 3.2.

Example 3.3 The infinite sequence of states constructed from Kripke structure defined in the example 3.2 can be:

Example 3.4 The computation tree constructed from Kripke structure defined in the example 3.2 can be:
The Computation Tree Logic (CTL) [2] is a temporal logic based on propositional logic with a discrete notation of time and only future modalities. It is a branching temporal logic which is used for model checking by Clarke and Emerson [9]. CTL can describe specifications on both states and paths. To express specifications CTL offer a set of operators; there are two groups of these operators:

- Path quantifiers
  - $A$: forall to quantify over all paths
  - $E$: exists to quantify over some paths

- Temporal Operators
  - $X$: the ‘next’ operator
  - $F$: to a property that holds ‘eventually’
  - $G$: to express a property that always hold
  - $U$: the ‘until’ operator

The visualization of semantics of some basic CTL formulae constructed with above operators [2]:

- $EF$ black
- $EG$ black
- $AF$ black
- $AG$ black
Definition 3.5 The CTL syntax is defined as:

\[ \phi, \psi ::= E(\phi U \psi) \mid A(\phi U \psi) \mid EX \phi \mid \phi \land \psi \mid \neg \phi \mid p \]

where \( p \) is an atomic proposition.

‘Atomic propositions’ are propositions expressing properties that can be evaluated on a state. Their exact formulation depends on what the properties we want to observe. E.g. for a weather system, the atomic propositions can be \texttt{isRaining}, \texttt{isSnowing}, etc. This would allow a quite sophisticated state property to be expressed. But the definition of Kripke assumes that states are structure-less. So technically speaking, atomic propositions are those propositions which are in the co-domain (range) of the Kripke’s interpretation function (the \( L \)-component).

Definition 3.6 The semantic of CTL operators is defined below. Let \( K \) be a Kripke structure and \( s \) be a state, \( \phi, \psi \) be CTL formulae.

The notation \( \Pi(s) \) means the set of all paths starting from the state \( s \). For a path \( \pi \), the notation \( \pi_i \) means the \( i \)-th state of \( \pi \).

The notation \( K, s \models \phi \) means \( \phi \) holds with \( K \) where

\[ K, s \models \phi = (\forall s_0 \in I : K, s_0 \models \phi) \]

The notation \( K, s \models \phi \) means that the CTL formula \( \phi \) ‘holds’ on the state \( s \), which is defined more precisely as below:

\[
\begin{align*}
K, s \models p & \quad \text{if } p \in L(s) \\
K, s \models \neg \phi & \quad \text{if not } K, s \models \phi \\
K, s \models \phi \lor \psi & \quad \text{if } K, s \models \phi \text{ or } K, s \models \psi \\
K, s \models EX \phi & \quad \text{if there exists a successor } t \text{ of } s \text{ such that } K, t \models \phi \\
K, s \models A(\phi U \psi) & \quad \text{if } \forall \pi \in \Pi(s) : (\exists i \geq 0, K, \pi_i \models \psi \text{ and } (\forall 0 \leq j < i, K, \pi_j \models \phi)) \\
K, s \models E(\phi U \psi) & \quad \text{if } \exists \pi \in \Pi(s) : (\exists i \geq 0, K, \pi_i \models \psi \text{ and } (\forall 0 \leq j < i, K, \pi_j \models \phi))
\end{align*}
\]

Other modalities like \( EF, AF, EG \) and \( AG \) can be derived from the modalities defined in Definition 3.5.
Definition 3.7

\[ K, s \models \varphi \land \psi \quad K, s \models \neg \varphi \lor \neg \psi \]

\[ \text{EF} \varphi \equiv E(\text{true} U \varphi) \]

\[ \text{AF} \varphi \equiv \neg \text{EG} \neg \varphi \]

\[ \text{EG} \varphi \equiv \neg \text{AF} \neg \varphi \]

\[ \text{AG} \varphi \equiv \neg \text{EX} \neg \varphi \]

CTL can be further extended into Bounded CTL[7], which add additional constructs to specify amount of steps that a specification is quantified over a path. The new constructs are:

\[ \text{EB} (i..j)(\varphi U \psi) \quad \text{AB} (i..j)(\varphi U \psi) \]

The operator B can be only bounded to path quantifiers. The semantic is defined below.

Definition 3.8 Assume that \( K \) is a Kripke structure, \( s \) is a state, and \( \varphi, \psi \) be CTL formulae. As before, \( \Pi(s) \) denote the set of all paths that branch out from the state \( s \).

\[ K, s \models \text{AB} (i..h) (\varphi U \psi) \quad \text{if} \quad \forall \pi \in \Pi(s) : (\exists l \leq i \leq h, K, \pi_i \models \psi \quad \text{and} \quad (\forall l \leq j \leq i, K, \pi_j \models \varphi)) \]

\[ K, s \models \text{EB} (i..h) (\varphi U \psi) \quad \text{if} \quad \exists \pi \in \Pi(s) : (\exists l \leq i \leq h, K, \pi_i \models \psi \quad \text{and} \quad (\forall l \leq j \leq i, K, \pi_j \models \varphi)) \]

3.3 UI model

A UI allows a user to interact with a system. It can be in different states so that user can perform different actions. The behavior of UI can be seen as sequences of states caused by series of actions performed by a user. When there is an action \( a \) that from some state \( s \) has with multiple next-states then we have a non-deterministic UI. This situation arises because of e.g. internal concurrency in the system, or abstract away some details on purpose like when we abstractly represent two deterministic actions with a single abstract action, the latter will (usually) become non-deterministic. At each state, the UI may hold different elements like widgets, contents, etc. Those elements are called UI elements. UI model is introduced to model UI behavior, and the UI elements involved along with their attributes.

Previously in Section 3.1, we have shown how to model a web-based UI with a Kripke structure. However, the model described by the Kripke structure in example 3.2 is very simple, it only captures the pages and the transitions between the pages. To allow more expressive model of UIs than what now possible with the combination of Kripke and (Bounded) CTL, we need a variant of Kripke structures.

We are inspired by the Kripke variant defined in [17], which is designed to describe hyperdocument systems. In [17], a hyperdocument is described as “an interactive document, providing a non-linear, dynamic analogue to the traditional notion of structured document”. At each state, the content of certain paragraphs is viewable. There are some links and buttons which link to other paragraphs. Conceptually, this is quite similar to a UI, click on links or buttons is the possible action provided by hyperdocument which will change the UI state. So the approach they proposed can also be used for UI, and the Kripke structure defined in [17] can also be applied to UI. Based on this extended Kripke structure, it is possible to describe browsing properties like whether some content is visible, whether some button is selectable, whether they are reachable and so on.
The example 3.2 maps ‘pages’ directly to the states of the Kripke. It actually changes the original semantic of states defined in the Kripke structure. The original semantic of states defined in the Kripke structure are just identifiers of the system states, the state names itself does not mean anything. In example 3.2, the semantic of states defined in the Kripke structure implies that at certain state, the relevant content is viewable. For instance, at the home state, it supposes that home page content is visible. Rather than using a mere Kripke structure, [17] suggests a slightly different variant of Kripke structure to represent a UI which allows to decouple the content visibility from the states defined in Kripke structure so that there is no changes on the original semantic of states.

Since a state in Kripke is structureless, it is impossible to express the concept of 'UI elements' which are 'visible' or 'selectable' in the state by encoding the information in the state itself. This has to be encoded in the interpretation function \( L \). However, \( L \) only associates a state with propositions, which is not always convenient. So the way that [17] express this information is by introducing two slightly different interpretation functions. They use a function \( C \) to conveniently express which UI elements are visible in each state, and another function \( B \) to express which action-like UI elements are selectable in each state. This adapted Kripke structure is defined below:

**Definition 3.9** Let \( K = (S, I, R, C, B) \) where

- For \( S, I, R \), they are \( K \)'s set of states, set of initial states, and set of transitions as defined before.
- \( C \) is a function that maps each state in \( K \) to a set of names; so, \( C : S \rightarrow \) a set of names. Each name is used to identify a UI element.
- \( B \) is a function that maps each transition in \( K \) to a name. That is, the type of \( B \) is \((S \times S) \rightarrow \) name. Each name abstracts UI element which triggers the transition.

Comparing with 3.1, the interpretation function \( L \) is dropped. Recall that it gives interpretation to each state, by associating it to a set of atomic propositions.

The role of \( L \) is now replaced by \( C \) and \( B \). Specifically for the hyperdocument, [17] only has two kinds of simple formulas (thus they are state formulas): content visibility and selectability. The function \( C \) is used to express the first; that is for a state \( s \), \( name \in C(s) \) means that a content abstractly represented by \( name \) is visible in the state \( s \). \( B(s, t) = name \) is used to express that a button or a link abstractly represented by \( name \) is thus ‘selectable’ at the state \( s \) (and furthermore leads to the state \( t \), but this additional information is irrelevant for the concept of selectability itself).

In formulas, [17] express visibility and selectability with a slightly indirect notation. They use ‘simple formulas’ of the form \( c.name \) and \( b.name \). The notation \( c.name \) means the content named ‘name’ is visible; similarly \( c.menu \) means content menu is visible. The notation \( b.name \) means that a button or a link named ‘name’ is selectable.

**Definition 3.10** The syntax of simple formula is:

\[
f ::= c.name \mid b.name
\]

Their exact meaning is defined below; unsurprisingly they are defined in terms of \( C \) and \( B \). Let \( K = (S, I, R, C, B) \) be a Kripke structure with the adapted structure as in Def. 3.9, and \( s \) be a state in \( K \):

\[
K, s \models c.name \equiv name \in C(s)
\]

\[
K, s \models b.name \equiv \exists t : (s, t) \in R \land B(s, t) = name
\]
So, $K, s \models c.name$ is just another way to express $name \in C(s)$. The notation is introduced probably to fit in the style common in CTL. Above definition 3.9 of Kripke cannot express properties of attributes like the number of links in a given page, e.g. as in Example 3.2. For analyzing usability-like properties, we will further extend the above definition later.

**Example 3.11** The example 3.2 without labeling can be adapted using Kripke structure with definition 3.9, as:

- $S = \{0, 1, 2\}$
- $I = \{0\}$
- $R = \{(0, 1), (1, 0), (2, 0), (1, 2)\}$
- $C = \{0 \mapsto \{\text{home}\}, 1 \mapsto \{\text{login}\}, 2 \mapsto \{\text{mailbox}\}\}$
- $B = \{\}$

The states are now represented by just numbers. The structure is depicted graphically below. In the picture, the set after each number $k$ is just the set of ‘names’ associated to $k$; so $C(k)$. This set tells which UI content in the actual web-based UI that would be visible in the actual states that correspond to the abstract state $k$.

The example above still maps ‘web pages’ directly to states in the Kripke structure. However, we can change the abstraction level of Kripke structure to demonstrate a more powerful model.

**Example 3.12** For instance, we can change the abstraction level to map states defined in Kripke structure to some internal states of web application like login successful or login failed.

- $S = \{0, 1, 2, 3\}$
- $I = \{0\}$
- $R = \{(0, 1), (1, 0), (0, 2), (2, 3), (2, 4), (4, 2), (3, 4)\}$
- $C = \{0 \mapsto \{\text{home}\}, 1 \mapsto \{\text{loginFail}\}, 2 \mapsto \{\text{loginOK}\}, 3 \mapsto \{\text{mailbox}\}, 4 \mapsto \{\text{home}\}\}$
- $B = \{\}$
In the above model, state 1 corresponds to login by an invalid data and state 2 to login by a valid data. The model is more refined, as it can express that the mailbox content is only visible after we login with valid data. Notice that there is no longer a 1-to-1 mapping between the 'web pages' and the model's states.

In definition 3.9, the function $B$ labels each transition with a name. The name expresses a button or a link which can trigger the transition bounded by function $B$. In above example, if there is button 'submit' that trigger transition $(0,1)$ and transition $(0, 2)$, the function $B$ should be defined as:

$$B = \{(0, 1) \mapsto \text{submit}, (0, 2) \mapsto \text{submit}, \ldots\}$$

### 3.3.1 Web based UI composition

A web based UI consist of one or more web pages. At beginning of this chapter, we stated that the texts, widgets and other elements are UI element. UI elements can be nested too (with each UI element can only has one parent). So in the end, a web page can be represented as a tree structure. Each node in the tree is a UI element, abstractly represented by its name.

**Example 3.13** Consider is a web-based UI containing a page with named 'homePage' with 4 widget: a menu named 'menu', a box named 'box' for displaying some texts, a link named 'homeLink' that simply point back to the home page, and a link named 'loginLink' that points to login page. The links sit however inside the menu. The tree structure of the UI elements belonging to the home page looks would like this:

```
home
  menu
    homeLink
  box
    loginLink
```

We call this tree structure UI structure tree. We can define a web page also becomes a UI element because we can also view a web page as a UI element, and it thus represented by a UI structure tree. In this thesis we will restrict to the following sort of UI elements: page, box, button, checkbox, dropdownbox, flash, image, inputbox, link, list and radio button. A Box can be used to allocate
texts, group several widgets together. It maps to div, label, span tags in HTML. Other elements just directly maps to the corresponding tag in HTML.

There may be some interesting attributes that we want to analyze for each UI element, we call those attributes as UI attributes. Different widgets may contains different UI attributes. In this thesis we limit ourselves to the following attributes.

<table>
<thead>
<tr>
<th>UI element</th>
<th>UI attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>page</td>
<td>url</td>
</tr>
<tr>
<td>box</td>
<td>width, height, positionX, positionY, content, isVisible</td>
</tr>
<tr>
<td>button</td>
<td>width, height, positionX, positionY, content, enabled, isVisible</td>
</tr>
<tr>
<td>image</td>
<td>width, height, positionX, positionY, alt, src, isVisible</td>
</tr>
<tr>
<td>link</td>
<td>width, height, positionX, positionY, content, enabled, isVisible</td>
</tr>
<tr>
<td>list</td>
<td>width, height, positionX, positionY, enabled, nrOfElements, isVisible</td>
</tr>
<tr>
<td>checkbox</td>
<td>width, height, positionX, positionY, enabled, nrOfOptions, isVisible</td>
</tr>
<tr>
<td>dropdownbox</td>
<td>width, height, positionX, positionY, nrOfOptions, isVisible</td>
</tr>
<tr>
<td>inputbox</td>
<td>width, height, positionX, positionY, content, enabled, isVisible</td>
</tr>
<tr>
<td>flash</td>
<td>width, height, positionX, positionY, src, isVisible</td>
</tr>
<tr>
<td>radio</td>
<td>width, height, positionX, positionY, nrOfOptions, isVisible</td>
</tr>
</tbody>
</table>

3.3.2 Representing web based UI with UI model

With Kripke structure defined as definition 3.9 and extension of simple formula defined in definition 3.10, we can model and describe the visibility and reachability of a UI element. However we cannot yet express tree structured UI elements, nor can we express UI attributes. So we need to extend the structure a bit.

Recall definition 3.9, function $C$ maps a state to a set of names. Instead of using it to only express e.g. content visibility for each state, we extend $C$ to map a state to a UI element. Each UI element consists of a unique identifier, a set of UI attributes and a set of UI elements. So the type of $C$ now is:

$$C : S \rightarrow UIelement$$

where:

- type $UIelement = id \times (set \ of \ attribute) \times (set \ of \ UIelement)$ where id is unique
- type $attribute = name \times value$ where pair of name and value is unique in set of attribute

If $P$ is a UI element; that is if $P = (id, A, components)$ is a UI element, we will write $attr(P)$ to denote the set of $P$'s attributes. That is:

$$attr(P) = A$$

We write $dom(attr(P))$ to denote the set of names which are ‘defined’ in $attr(P)$. That is:

$$dom(attr(P)) = \{ n \mid (\exists value. (n, value) \in attr(P)) \}$$

We will write $elem(P)$ to denote the set of UI elements which are ‘defined’ in this UI element which are components of $P$. That is:

$$elem(P) = components$$

The notation $P(name)$ which usually means function application can be overloaded to mean the set of all values in $attr(P)$ associated to $n$. So, for $n \in dom(attr(P))$:

$$P(name) = \{ value \mid (n, value) \in attr(P) \}$$

Further, $P(id)$ is defined to mean the UI element in $elem(P)$ with id ‘id’. Since the UI element is constructed recursively, the $P(id)$ is defined as:
if $P = (id, -, -)$ then $P(id) = P$

if $P = (id, \text{attributes}, \emptyset)$

$(id, \text{attributes}, \emptyset)(id') = \emptyset$ where $id \neq id'$

otherwise $P(id) = \bigcup_{Q \in \text{elem}(P)} Q(id)$

Since the id is unique, the set returned by $P(id)$ always contains one UI element. So we overload $P(id)$ by

$P(id) = Q$ where \{$Q\} = P(id)$

So if $s$ is a state in $S$, $\text{dom(attr}(C(s)))$ returns all names of the attributes of the UI element visible in the state $s$ and $C(s)(\text{name})$ returns the values of the attribute $\text{name}$. $C(s)(id)$ returns the UI element associated with the identity $id$ which are visible in the state $s$. To get certain UI attribute from UI element, we can write $C(s, id, \text{name})$ where $C(s, id, \text{name}) = C(s)(\text{id})(\text{name})$.

In definition 3.9, function $B$ maps a transition $(s, t)$ to the id of the links or buttons that are selectable in the state $s$ and causes the transition $(s, t)$. Links and buttons can be treated as UI element. The widgets visibility can be defined in UI element as a UI attribute inside function $C$. In UI model, we do not need function $B$ anymore.

Recall that definition 3.10, simple formulas have the following form:

$f ::= c.\text{name} \mid b.\text{name}$

Also recall that in this syntax $c.\text{name}$ and $b.\text{name}$ are predicates (they ‘returns’ a boolean value). We will now extend the syntax of $c.\text{name}$ and remove $b.\text{name}$; but in the new syntax $c.\text{name}$ are no longer predicates. $c.\text{name}$ will mean the value of the attribute $\text{name}$ (on the given state and UI element).

The new syntax for simple formulas is:

$f ::= t = t \mid t < t \mid t > t$ (simple formula)

$t ::= c.id.\text{name} \mid \text{value}$ (term)

Their meaning is defined below.

**Definition 3.15**

Let $[t]$, mean the meaning of term $t$ on a state $s$:

$$[c.id.\text{name}]_s = \begin{cases} C(s, id, \text{name}), & \text{if } name \in \text{dom(attr}(C(s, id))) \\ \bot, & \text{otherwise} \end{cases}$$

**Definition 3.16**

The meaning of a state formula $t = u$ (whether two terms are equal to each other) on a given state $s$ is:

$$s \models t = u = \begin{cases} [t]_s = [u]_s, & \text{if both } [t]_s \text{ and } [u]_s \text{ are } \bot \\ F, & \text{otherwise} \end{cases}$$

The meaning of $<$ and $>$ is analogous.
**Definition 3.17** Finally now we can define what a 'UI model' is. A UI model is how we in our framework models an actual UI. It is a variation of Kripke structure. Formally, UI model \( M \) is defined as a Kripke-like structure \((S, I, R, C)\) where:

- \( S \) is a finite set of states. These are the states of \( M \).
- \( I \) is the set of \( M \)'s initial states; \( I \subseteq S \).
- \( R : \text{set of } S \times S \) is the set of \( M \)'s transitions.
- \( C \) is a functions which map each state in \( M \) to a UI element. Each UI element consists of a unique identifier, a set of UI attributes and a set of UI elements.

\[ \square \]

**Example 3.18** With definition 3.17, the function \( C \) in example 3.12 should be adapted to using UI element with unique identifier for expressing this UI element is visible for at certain state. And definition 3.17 also can support the link attributes defined in example 3.2.

- \( S = \{0, 1, 2, 3, 4\} \)
- \( I = \{0\} \)
- \( R = \{(0, 1), (1, 0), (0, 2), (2, 3), (2, 4), (4, 2), (3, 4)\} \)
- \( C = \begin{cases} 0 & \mapsto (\text{home}, (\text{links}, 1), (\{}), 1 & \mapsto (\text{loginFail}, (\text{links}, 2), (\{}), 2 & \mapsto (\text{loginOK}, (\text{links}, 2), (\{}), 3 & \mapsto (\text{mailbox}, (\text{links}, 1), (\{}), 4 & \mapsto (\text{home}, (\text{links}, 1), (\{})) \end{cases} \)

![UI model abstraction level diagram]

**UI model abstraction level**

Examples 3.11 and 3.12 show that we can use Kripke structures as models at the different abstraction level. In general, the UI engineers should decide what are the proper abstraction level for them. Generally, we can speak of two different levels of abstraction, namely page, internal-state:
Definition 3.19  
• Page level abstraction. A UI model is at this level of abstraction if it directly maps pages to states. That is, each page is mapped to one state. The transitions describe the links or buttons that direct the user to other pages. The model in Example 3.11 is of this level. The labelling reveals no other information than the names of the pages.

• Internal-state level abstraction. A UI model is at this level if it reflects internal states of pages if it can give us some information about the internal states of the modeled application. E.g. if a certain page has two different internal states, they can be mapped into two different states in the UI model. We also can add information about internal states is by encoding them in the labeling. The model in 3.12 is of this level.

Example 3.20  
The internal state level abstraction can be very different. For instance, based on example 3.11, the example 3.18 distinguish the login page with two internal states: success and failed. But if we change the abstraction level based on actions, which means each actions means cause a different states, we will get a different UI model.

Based on example 3.11, suppose there is link 'login' at the home page that links to the login page. At the login page, the user has to type in his username in the inputbox 'username' and his password in the inputbox 'password'. He can click the submit button 'ok' that will direct him to the mailbox page, or to the error page (if he typed in invalid data). The error page only contains a link 'home' that links back to the home page. At the mailbox page, there is also a link 'home' that links to the home page.

If we distinguish internal states according users actions like after users typing in their username, we say login page is in different state, the UI model could be defined as:

- \( S = \{0, 1, 2, 3, 4, 5\} \)
- \( I = \{0\} \)
- \( R = \{(0,1), (1,0), (1,2), (1,4), (2,0), (2,3), (2,4), (3,0), (3,4), (3,5), (4,0), (5,0)\} \)
- \( C = \{0 \mapsto (\text{home}, (\text{links}, 1), \{\}), 1 \mapsto (\text{login}, (\text{links}, 2), \{\}), 2 \mapsto (\text{login}, (\text{links}, 2), \{\}), 3 \mapsto (\text{login}, (\text{links}, 2), \{\}), 4 \mapsto (\text{error}, (\text{links}, 2), \{\}), 5 \mapsto (\text{mailbox}, (\text{links}, 1), \{\})\} \)

To make above example more clear about which action causes which state, we explicitly give the mapping between transitions and actions below:

<table>
<thead>
<tr>
<th>Transition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,1)</td>
<td>click login link</td>
</tr>
<tr>
<td>(1,0)</td>
<td>click home link</td>
</tr>
<tr>
<td>(1,2)</td>
<td>type in username</td>
</tr>
<tr>
<td>(1,4)</td>
<td>click submit button</td>
</tr>
<tr>
<td>(2,3)</td>
<td>type in password</td>
</tr>
<tr>
<td>(2,4)</td>
<td>click submit button</td>
</tr>
<tr>
<td>(2,0)</td>
<td>click home link</td>
</tr>
<tr>
<td>(3,0)</td>
<td>click home link</td>
</tr>
<tr>
<td>(3,4)</td>
<td>click submit button</td>
</tr>
<tr>
<td>(3,5)</td>
<td>click submit button</td>
</tr>
<tr>
<td>(4,0)</td>
<td>click home link</td>
</tr>
<tr>
<td>(5,0)</td>
<td>click home link</td>
</tr>
</tbody>
</table>
Chapter 4

UI task and UI category

The UI model is introduced to describe the entire web based UI. However, a web based UI can contain a huge amount of web pages. Suppose we model a web based UI at the page level, and there are $n$ pages. Suppose that we have users that can login in $m$ different roles, then the corresponding UI model may contain $m \times n$ states. The situation is getting worse when we try to model the UI at the action level. Suppose that on average there are $s$ actions per page, then the UI model may have $m \times n \times s$ number of states. Although the model checker can handle the huge amount of states, the performance of the model checker is decreased as the amount of states increased. [6][14] discuss several state space reduction techniques which aims to reduce resource usage for verifying model with huge amount of states. On the another hand, the problem can be solved by reducing analysis scope. However, reducing the scope may mean that our verification is only partial, so we have to be reasonable in doing so. In this chapter, we introduce two kinds of scope: UI task and UI category.

4.1 UI task

When a user visits a web application, he tends to have certain purposes, e.g. getting certain information, buying stuff (if the application is an online shop), etc. To achieve a goal, the user has to perform a sequence of actions by interacting with the application. The goal and the sequence of actions together describe a task. There are several ways to model tasks introduced in the past; we are particularly inspired a model approach called GOMS.

4.1.1 GOMS

A GOMS model [12] is a description of the “how to do it” knowledge that is required by an application in order to get intended tasks accomplished. The GOMS itself is an abbreviation of Goals, Operators, Methods and Selections rules. A task in the GOMS is composed by those four elements. To be more precise, a task has a specified goal and a set of methods which can accomplish the goal. Each method contains a sequence of steps/actions that the users have to perform which are defined as operators. Furthermore, inside a method, several sub-goals may be required to be finished. When there are more than one methods to accomplish a goal, selection rules are defined to determine which method should be chosen.

An operator represents an action that a user or application need to perform. Totally, there have 3 kinds of operators.

- **External operators** are actions where the application interacts with its outside environment, e.g. the users.
- **Mental operators** are internal actions performed by the users. The mental operators can be classified into three groups.
  - **Standard primitive mental operators** are actions like making decision.
- Memory storage and retrieval operators are actions that reflect the distinctions between long term memory and short term memory like recall, retain, forget information.
- Analyst-defined mental operators are actions which require users to do analysis before performing actions like determining a string used for find command
- Primitive and high-level operators are actions which are defined at the high level for analysis purpose. It can be decomposed into a sequence of lower-level operators. For instance, a typical low-level operator can be the action like pressing a button. A high-level operator can be the action like logging into system which can be decomposed into a sequence of low-level operators.

A method is composed by a sequence of steps. A step is either an operator or a sub-goal.

A selection rule is a flow control for determining the method used to accomplish a goal. For instance, suppose that in an application there are two ways to select text. If the text is a word, we can move the cursor to the middle of the word, then we double click our mouse button, then we verify that the correct text is selected. However, if the text is not a word (e.g. a sentence), we first move the cursor to the beginning of the text, then we press the mouse button down, we locate the end of the text, then move mouse to the end of the text, then we verify that the correct text is selected, and finally we release the button.

The above selection rule can be described in GOMS like this:

Selection rule set for goal: select text
  If text-is word, then
    Accomplish goal: select word
  If text-is arbitrary, then
    Accomplish goal: select arbitrary text
    Return with goal accomplished

The select word and select arbitrary are two subgoals:

Method for goal: select \{object\}
  Step 1. Locate beginning of \{object\}
  Step 2. Move cursor to beginning of \{object\}
  Step 3. Double-click mouse button
  Step 4. Verify that correct text is selected
  Step 5. Return with goal accomplished

Method for goal: select arbitrary \{object\}
  Step 1. Locate beginning of \{object\}
  Step 2. Move cursor to beginning of \{object\}
  Step 3. Press mouse button down
  Step 4. Locate end of text
  Step 5. Move cursor to end of text
  Step 6. Verify that correct text is selected
  Step 7. Release mouse button
  Step 8. Return with goal accomplished

4.1.2 UI task

GOMS models are task based model which is procedural. We also can define the task based on the UI model. Our purpose rendering task concept in the UI model is reducing the size of UI model. The UI task only needs to specify the states and transitions concerned by a certain task. By using a UI task instead of a UI model, we can limit the analysis of the UI model to the part that is relevant to the given task.
Definition 4.1 A UI task will be represented by a Finite State Machine $K$, which is a tuple $(S, I, R)$ where:

- $S$ is a finite set of states.
- $I$ is a set of initial states such that $I \subseteq S$.
- $R : \text{set of } S \times S$ is the set of $K$’s transitions.

□

The only difference with UI model is that UI task does not contain $L$.

Example 4.2 Based on example 3.20, suppose we want to add a task: compose a mail. After a successful login, and reaching state 5, the user needs to click compose mail link, then he needs to fill in the mail receiver and the mail body fields, and click the send button in the end. When clicking the send mail button, if one of the mail receiver and mail body is empty, the error is shown.

We also add a group of privacy pages. There is a link privacy at the home page which links to the privacy center page. There are three pages in the privacy group: privacy center page, privacy overview page, privacy FAQ page, and they can link to each other.

We give the simplified UI model below, the full UI model can be found at appendix 11.1:

- $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\}$
- $I = \{0\}$
- $R = \{(0, 1), (0, 11), (1, 0), (1, 2), (1, 4), (2, 0), (2, 3), (2, 4), (3, 0), (3, 4), (4, 0), (5, 0), (5, 6), (6, 7), (6, 10), (6, 9), (7, 6), (7, 10), (7, 8), (8, 7), (8, 5), (9, 8), (9, 10), (9, 6), (10, 6), (11, 12), (11, 13), (11, 14), (12, 11), (12, 13), (12, 14), (13, 11), (13, 12), (13, 14), (14, 11), (14, 12), (14, 13)\}$
- $C = \{0 \rightarrow (id : \text{home}), 1 \rightarrow (id : \text{login}), 2 \rightarrow (id : \text{login}), 3 \rightarrow (id : \text{login}), 4 \rightarrow (id : \text{error}), 5 \rightarrow (id : \text{mailboxIndex}), 6 \rightarrow (id : \text{mailboxNewMail}), 7 \rightarrow (id : \text{mailboxNewMailBodyEmpty}), 8 \rightarrow (id : \text{mailboxNewMailReceiverBodyFilled}), 9 \rightarrow (id : \text{mailboxNewMailReceiverEmpty}), 10 \rightarrow (id : \text{error}), 11 \rightarrow (id : \text{privacyCenter}), 12 \rightarrow (id : \text{privacyOverview}), 13 \rightarrow (id : \text{privacyPolicy}), 14 \rightarrow (id : \text{privacyFAQ})\}
Based on above UI mode, the compose mail task can be defined as:

- \( S = \{5, 6, 7, 8, 9, 10\} \)
- \( I = \{5\} \)
- \( R = \{(5, 6), (6, 7), (6, 10), (6, 9), (7, 6), (7, 10), (7, 8), (8, 7), (8, 5), (9, 8), (9, 10), (9, 6), (10, 6)\} \)
if $M$ is the UI model and $T$ is the task model, by taking the ‘intersection’ of the $M$ and $T$ to obtain a new, smaller model $M'$. This $M'$ will take over the $C$ from $M$. By analyzing $M'$ instead of $M$, we reduce our analysis scope.

4.2 UI category

The UI category can be defined by grouping the UIs based on their purposes, e.g. ‘product pages’ only provide information about products. The UIs in the same category often have similar requirements e.g. in layout, style etc. Some UI specifications may only be applicable to the UIs from certain categories. So, for analyzing such a UI specification we do not need to take the full UI model into the analysis procedure.

Definition 4.3 Since a UI category only defines a set of UIs, it is not necessary to include transitions in its definition. So, a UI category will be represented simply by a set of states ($S$).

Example 4.4 With the example 4.4, all privacy pages can be grouped into privacy category as:

- $S = \{11, 12, 13, 14\}$

A UI category is just a set of states, is just a special case of UI task model, namely one with an empty set of transitions. Thus the UI category can be used to restrict the analysis scope too.
Chapter 5

UI specifications

The UI specifications describe the UIs in different aspects like functional aspects, style aspects, etc. Before designing a UI, the UI engineers need to analyze the UI requirements to get an overview about e.g. who will use the UI and how UI should look like. Based on this information, the UI engineers make UI specifications. When designing the UI, the UI engineers should ensure that the UIs satisfy those specifications. The specifications can be used to specify the properties with different concerns. In this section, we mainly discuss two concerns: usability and security.

5.1 UI specification for usability properties

Usability of a tool is the ease with which people can employ the tool in order to achieve a particular goal [10]. There are many guides [1], e.g. specifying the methodologies about how to measure usability. The standards for judging good usability still need to be defined by UI engineers, e.g. the response time can be very different based on different system. Although there are no concrete definitions about good usability, many experimental studies were carried out on web applications’ usability [13][18], leading to some aspects that can affect usability. In this section, we introduces some aspects and express them in CTL.

- Browsing
  Browsing is a basic features of the web application. The prerequisite for all interactions with web application is that users can browse web pages which provide information and services. Browsing web application can be represented as a sequence of web pages that user can visit. However, when there are many web pages, the relations between those web pages becomes complicated and untraceable, there may have some problems which make web application does not behave effectively that can facilitate users to accomplish their goals. They can be summarized into three usability aspects: reachability, dead end and browsing efficiency.

  - Reachability
    Reachability analysis examines whether a web page at least can be reached via a browsing path. If a web page can not be reached from any other web pages, this page is isolated. This is probably a mistake in the UI design, the UI engineers may forget to link this page with other pages in the UI design.

    The CTL formula for expressing that “The page p at least can be reached” can be defined as: $EF \ c.id = p$

  - Dead end
    Dead end means once a user visits a web page, there is no way that user can leave this web page. In this case, user can escape from current web page by closing browser or clicking the back button on the browser.

    If we look at internal state of a web page, the dead end also means once a user get into a
state of a web page, there is no way that user can go to a different state of this web page. This situation normally occurs in the web application which is written with the technologies like AJAX, etc. Those technologies enable the web application to send request to a server and refresh the current web page contents without changing the browser’s url. So users just interact with the web application, get into different states without actual web page changing. In this case, clicking on back button from browser can not make the web application go back to previous state.

Although detecting dead end may not be always necessary, in our framework, we provide a way to use CTL formula to represent such a dead end situation. So that UI engineers can detect dead end depends on real situation.

That a page $p$ is not a dead end can be expressed by the CTL formula:

$EF((c.id = p) \land (Ec.id \neq p))$

- **Browsing efficiency**

A web application may contains a huge amount of information. Some information may be very important that UI engineers expect those information can be easily accessed, e.g. it is common that an online shop page should be easily accessed by users. The way to access a web page can be represented into a sequence of pages that visitors need to pass through. The less pages in this sequence means this page can be accessed easier.

The statement: "a page $p$ is reachable from initial state within $k$ steps" can be defined with Bounded CTL formula: $EBF 0..k .c.id = p$

- **UI attributes**

In this section, we give two examples of UI attributes which may affect web application’s usability.

- **Amount of words**

A excessive long and dense web page requires the users to scroll the web page a lot and make the web page content hard to read which may frustrates the users and let them to skip reading the rest parts of the page. The amount of words in a web page can affect page length and page density.

The CTL formula for constraining the amount of words less than $k$ at all page be defined as: $AG(c.nrOfWords < k)$

The length of UI element may also affects the usability, e.g. an excessively long link may be hard to read. In some cases, constraining the amount of words for each UI element can constraint the UI element length like constraining the amount of words for the links, etc.

- **Amount of links**

Similar with the amount of words, the amount of links also affect page density.

The CTL formula for Constraining the amount of links to be less than $k$ at current page can be defined as: $AG(c.nrOfLinks < k)$

5.2 **UI specification for security**

Security issue is a critical issue for a web application. Authentication and authorization are two important components that provide security for a web application. Authentication concerns about how users can login their own identity in the application securely. For instance, the most used authentication solution is to ask users to enter their user name and password and authenticate them at server side. If users successfully authenticate themselves, they are given a role. Authorization uses the role to decide if such user has enough permission to perform an action or access certain resource.
In web application, normally, there is a web page that provides authentication to users. If users successfully authenticate themselves, the system updates their roles and redirects user to a successful-login-page otherwise users may have to login again or are sent to a failed-login-page. If successful authentication is associated with a single state \( n \) in UI model, the authentication can be looked like a role transition after users pass state \( n \). If the web pages which associated with the states that appear after passing state \( n \) and all browsing paths that reach those web pages must pass the state \( n \), then those web pages are secured by state \( n \). Below is an example that demonstrates how the UI model represents authentication and what a state is secured by another state is.

Example 5.1 For instance, in example 3.12, we suppose that there are two roles for users: guest and login. After state 2, the users successful authenticate themselves, their role are transformed from guest to login. The page mailbox associated with state 3 is secured by state 2.

The authorization in a web application can be reflected by both the UI and the server. At the UI side, the authorization is represented by if a UI element is visible or enabled. At the server side, the authorization is represented by if the incoming request has enough permission to request data or update records, etc. In this thesis, we are focusing on analysis of authorization at the UI side. To carry out such analysis, it requires UI engineers to collect information about users’ roles, which UI elements are visible or enabled for which users’ roles. Further, it also requires the UI model contains the role information so that the analysis can be carried out based on different roles. Below is an more complicated example to show how we can represent security issue in CTL formula:

Example 5.2 The UI model proposed with definition 3.17 is able to contain he role information by representing each role as different state. For instance, in example 3.12, if state 2 means users successful to login with role \( A \), then all the states appear after passing state 2 are associated with role \( A \). Similarly, we can model a new role \( B \) as following diagram:

```
0;{home}         1;{loginFail}
start
2;{loginWithA}   3;{mailbox}
4;{homeWithA}
5;{loginWithB}   6;{homeWithB}

7;{admin}
```

Now, we can state that admin should always be accessed after user successful login as role \( B \) by CTL formula: 
\[ AG(\neg c.id = 'admin' \ U c.id = 'loginWithB') \].
Chapter 6

UI Verification

6.1 Model checker

Expected system properties, e.g. over its behaviors and looks, are defined in various kinds of specifications. Subsequently we will want to check if they are met. This can be done in several different ways: simulation, testing and model checking. Our framework support the last two ways.

Simulation requires an (executable) model of the system. It triggers the model to do executions, and the specifications are validated against these executions.

Testing is performed on a real system: we trigger it to do executions, and validate the specifications on these executions. Testing inherently cannot cover all system behaviors. Simulation based on a complex model has the same problem.

Model checking differs from testing and simulation in that it is fully exhaustive. It is also completely automatic, see e.g. Clarke & Emmerson’s algorithm [5][8]. Several tools to do model checking are available, e.g. SMV [4] and SPIN [11]. Since model checking can cover all possible situations, it can be used to discover very subtle errors hidden in system. The automation means that the whole procedure is efficient and low cost.

Model checking works on a model of the system. The model is written in a modeling language accepted by the model checking tool, and the system specifications should be represented in a certain formalism —usually it is temporal logic. A model checking tool exhaustively explores all possible states and executions of the model and checks if those executions satisfy the given specifications.

Furthermore, if the model checking tool finds that a specification does not hold, it can construct a counter example (an execution that leads to the violation of the specification).

6.2 SMV

Temporal logics can be classified into two classes: branching-time and linear time. In chapter 3, we discuss branching-time temporal logic CTL. A linear time temporal logic defines an execution of a system as a sequence of state transitions, whereas a branching time logic defines it as a tree of state transitions. This different views make a different in the kind of properties that can be expressed, and furthermore also make a (big) different in how model checking works. There are two mainly used
model checking tools: SPIN [11] and SMV [4]. SPIN is for model checking a linear temporal logic, and SMV for a branching time temporal logic. We will use SMV. There are many adapted versions of SMV with different algorithms. But all of them use same SMV language as input. The temporal logic supported by SMV is CTL.

The syntax details of the SMV program can be found in [4]. In this section, we will briefly describe the structure of a executable SMV program.

6.2.1 Variable declaration section

In the SMV program, Variable declaration section declares the variables and each variable defined as:

Variable : (name \times type)

For instance, we can declare a boolean type variable ‘request’ by:

request : boolean;

A variable may also take an enumerated type, which can be specified explicitly e.g. like this:

state : \{ready, busy\} ;

6.2.2 ASSIGN section

The ASSIGN section initializes the variables and specifies how the variables’ values change from the current state to the next one.

\begin{verbatim}
ASSIGN
    init (state) := ready;
    init (request) := \{0, 1\};
    next (request) := case
        1 : \{1, 0\};
    esac;
    next (state) := case
        state = ready & request = 1 : busy;
        1 : ready;
    esac;
\end{verbatim}

The code above will initialize the variable state to the value ready and the variable request to be either 0 or 1 non-deterministically. Furthermore it specifies that the value of request at next state will be either 0 or 1 non-deterministically and when the value of state is currently equal to ready and request is true, then in the next transition the value of state would be set to busy. Otherwise (the case 1 matches any situation, and is used for setting default next-state), its value will be ready.

6.2.3 Specification declaration section

The SMV program accepts CTL formula directly e.g:

\begin{verbatim}
SPEC AG(request \rightarrow AF(state = busy))
\end{verbatim}

6.2.4 Module declaration section

In the SMV program, the module is defined as:

\begin{verbatim}
Module : (Variable declaration section, ASSIGN section, Specification declaration section)
\end{verbatim}

All three sections are not compulsory.

A executable SMV program can have more than one modules, but it must have one Module declared as main. One module can declare a variable with type referring to another module, e.g:
When a module $A$ is referred as a type by a variable defined in another module $B$, the variable value defined in module $A$ can be retrieved by assembling all variable names from the referring hierarchy, e.g. $c$.center.x in above example.

6.3 Mapping UI model and UI specification

In definition 3.17, we defined what a 'UI model' is. In this section, we will explain how to represent UI model in the SMV program.

Let UI model $M$ is defined as a Kripke-like structure $(S, I, R, C)$, $S, I, R$ can be represented by the SMV program straight forward:

- $S$ can be represented by a variable with enumerated type
- $I$ can be represented by variables initialization in ASSIGN section
- $R$ can be represented by next statements in ASSIGN section

The function $C$ is recursively defined which reflects the UI elements hierarchy. We can directly represent $C$ in the SMV program by defining module for each UI element with variables which represent UI attributes, declaring variables with type referring another module to represent the UI elements hierarchy. A extra attribute $isVisible$ is introduce for indicating if the UI element is visible. This approach is however not really practical. We will discuss this later; let us here first see how this approach works.

**Example 6.1** Consider is a web-based UI containing a page with id 'home' with 3 widgets: a menu with id 'menu', a link with id 'homeLink' that simply point back to the home page, and a link with id 'loginLink' that points to login page. The links sit however inside the menu. The tree structure of the UI elements belonging to the home page looks would like this:

```
home
  menu
    homeLink
    loginLink
```

For simplicity, in this example, we only model $words$ attribute which represents amount of words
for each UI element that 'home' page has 100 words, 'menu' element has 30 words, 'homeLink' has 2 words, loginLink has 2 words.

The 'home' page can be directly represented in the SMV program as:

```
MODULE loginLink
  VAR id : {loginLink, undefined};
  words : 0..50;
  isVisible : boolean;
  ASSIGN
    init(id) := loginLink;
    init(words) := 2;
    init(isVisible) := 1;

MODULE homeLink
  VAR id : {homeLink, undefined};
  words : 0..50;
  isVisible : boolean;
  ASSIGN
    init(id) := homeLink;
    init(words) := 2;
    init(isVisible) := 1;

MODULE menu
  VAR id : {menu, undefined};
  words : 0..50;
  homeLink : homeLink;
  loginLink : loginLink;
  isVisible : boolean;
  ASSIGN
    init(id) := menu;
    init(words) := 30;
    init(isVisible) := 1;

MODULE home
  VAR id : {home, undefined};
  words : 0..150;
  menu : menu;
  isVisible : boolean;
  ASSIGN
    init(id) := home;
    init(words) := 100;
    init(isVisible) := 1;

MODULE main
  VAR home : home;
```

With the definition 3.15, Using terms like c.$id$.name we can, in our specifications, refer to the value of a particular attribute of a particular UI element, e.g. c.homeLink.words. With above SMV program, However the notation requires the complete 'path' ($id_0..id_n$) to the element to be specified, e.g
home.menu.homeLink.words. So when we map UI specifications into SMV program, we need find out the UI elements’ hierarchies.

Although the way that maps UI model and UI specification into the SMV program works, there still have two things can be improved:

- A UI module may contain many UI elements, declaring a module for each UI element may results a big SMV program.
- The SMV model checker produces model checking results with CTL formulas defined in the SMV program. If we have a very big UI elements’ hierarchies, keeping UI elements’ hierarchies in CTL formulas will let the SMV model checker produces very long result messages which are not understandable.

For the first improvement, we create a common module called UIelement which can be reused by all UI elements.

For the second improvement, we decide to transform the UI model into a flattened UI model before mapping it into the SMV program.

A flattened UI model is obtained by collapsing the hierarchy of the UI elements into a maximum two levels structure, and then adding some extra calculated statistics attributes called UI abstract attributes that that captures back some of the original hierarchy information.

For each sub UI element, in the flattened UI model, it will be directly linked to the root UI element. Then we add UI abstract attributes to each the sub-element. We defined twelve UI abstract attributes: nrOfUIElements, nrOfBoxes, nrOfButtons, nrOfImages, nrOfLinks, nrOfLists, nrOfCheckBoxes, nrOfDropdownBoxs, nrOfInputBoxs, nrOfFlashes, nrOfRadioes and nrOfWords.

Example 6.2 The SMV program for the example 6.1 can be adapted into:

```smv
MODULE UIelement
  VAR id: {loginLink, homeLink, menu, home, undefined};
  words: 0..10;
  nrOfWords: 0..100;
  isVisible: boolean;

MODULE main
  VAR home: UIelement;
  menu: UIelement;
  homeLink: UIelement;
  loginLink: UIelement;

ASSIGN:
  init (home.id) := home;
  init (home.words) := 1;
  init (home.isVisible) := 1;
  init (home.nrOfWords) := 6;

  init (menu.id) := menu;
  init (menu.words) := 1;
  init (menu.isVisible) := 1;
  init (menu.nrOfWords) := 5;

  init (homeLink.id) := homeLink;
  init (homeLink.words) := 1;
  init (homeLink.isVisible) := 1;
  init (homeLink.nrOfWords) := 2;
```
Example 6.3 For showing a complete example, let the UI model $M$ is defined as:

- $S = \{0, 1\}$
- $I = \{0\}$
- $R = \{(0, 1), (0, 0), (1, 0)\}$
- $C = \{0 \mapsto (\text{home}, (\text{words}, 1)), (\text{menu}, (\text{words}, 1)), ((\text{homeLink}, (\text{words}, 2)), (\text{loginLink}, (\text{words}, 2)), ([[]])), 1 \mapsto (\text{login}, (\text{words}, 1)), (\text{homeLink}, (\text{words}, 3)), [[]])\}$

The SMV program represents this UI model as:

```plaintext
MODULE UIelement
VAR id : \{loginLink, homeLink, menu, home, undefined\};
words : 0..10;
nrOfWords : 0..100;
isVisible : boolean;

MODULE main
VAR state : \{0, 1\}
  home : UIelement;
  menu : UIelement;
  homeLink : UIelement;
  loginLink : UIelement;
  login : UIelement;

ASSIGN:
  init (state) := 0;

  init (home.id) := home;
  init (home.words) := 1;
  init (home.isVisible) := 1;
  init (home.nrOfWords) := 6;

  init (menu.id) := menu;
  init (menu.words) := 1;
  init (menu.isVisible) := 1;
  init (menu.nrOfWords) := 5;

  init (homeLink.id) := homeLink;
  init (homeLink.words) := 1;
  init (homeLink.isVisible) := 1;
  init (homeLink.nrOfWords) := 2;

  init (loginLink.id) := loginLink;
  init (loginLink.words) := 1;
  init (loginLink.isVisible) := 1;
```

This example only exposes $nrOfWords$ attribute. Now, the notations like $c.id.name$ in the UI specifications can be directly represented in the SMV program with $id.name$. The flatten UI model above now has only two levels.
init (loginLink.nrOfWords) := 2;
init (login.id) := login;
init (login.words) := 1;
init (login.isVisible) := 0;
init (login.nrOfWords) := 4;

next (state) :=
case
  state=0 : {0, 1};
  1:0;
esac;

next (home.isVisible) :=
case
  next (state)=0 : {1};
  1:0;
esac;

next (menu.isVisible) :=
case
  next (state)=0 : {1};
  1:0;
esac;

next (homeLink.isVisible) :=
case
  next (state)=0 : {1};
  1:0;
esac;

next (loginLink.isVisible) :=
case
  next (state)=0 : {1};
  1:0;
esac;

next (login.isVisible) :=
case
  next (state)=0 : {0};
  1:1;
esac;

next (loginLink.words) :=
case
  next (state)=0 : {2};
  1:3;
esac;
Chapter 7

UI testing

The UI testing is manually carried out by the UI engineers to inspect if UI implementations satisfy their UI specifications. In this thesis, we want to automate the UI testing.

We generate a sequences of the user actions from the UI model. These sequences are used to drive a testing tool called Selenium. Selenium in turns drives a web browser to interact with the target web application. The conditions that decide if the tests passes or fails are generated from the UI specifications.

Since the UI specifications are expressed in the CTL formulas. By definition, to check a CTL formula we need to evaluate it against all computation trees that a target application can generate. But usually there are infinitely many of them. So our approach is to just generate a finite number of those trees, and check the CTL formulas on those trees. This approach is not complete, but still sound. This implies that a violation we find will be an actual violation of the application.

However, we cannot use a computation tree directly to drive the tests, because we don’t always have the ability to roll-back the target application to its previous state. So we generate sequences rather than trees. After we retrieve all UI attributes from the real UIs for the sequences, we reconstruct the tree from the sequences. Then we further convert the reconstructed tree into a flattened UI model, and check the CTL formulas relying on the SMV model checker.

7.1 UI specification testing

To test the UI specifications, there are two questions that have to be answered:

- How to generate test cases and evaluate the UI specifications?
- How to test against real UIs?

7.1.1 Test cases generation and UI specifications evaluation

Suppose we have a UI to test. We are also given a (UI) model of the UI, and a bunch of (UI) specifications. The test case defines a set of sequences of user’s actions on a given UI. Each user’s action may causes the UI into a different state in the UI model. So, each sequence can be represented by a sequence of transitions in the UI model.

The UI specifications are expressed in CTL formulas. Recall that such a formula can be evaluated on computation trees as explained in Section 3.2. Because a computation tree represents a set of paths among all possible states and each path is a sequence of states which also formulates a sequence of transitions, a computation tree actually describes a set of sequences of user’s actions which is a test case.
Example 7.1 For instance, if we have a UI model:

From this UI model, we can generate a computation tree for testing purpose:

Each CTL formula describe the constraints for one or more UI attributes. To evaluate a CTL formula on the computation tree, we need to solve:

- Collect the UI attributes for each state
- Evaluate CTL formulas on the computation tree

Collect the UI attributes for each state

The way that drives the real UIs into different states and retrieves the UI attributes from the real UIs are described in section 7.1.2.

In this section, we mainly focus on what the paths we should use to drive the real UIs.

The straight forward solution is directly traversing the computation tree. But there is problem with traversing a tree. When we reach a node $x$ we may decide that we want to backtrack to its parent node and try a sibling $y$. In above example, when we stay at the state 3, to reach the state 4, we have to make the web application back to the state 2 first. But the state rollback is not always possible as mentioned in section 5.1.

To avoid the state rollback, instead of generating the computation tree, we first generate a set of paths from the initial state which is actually a restricted computation tree that each node has only one child. So the test case now is a set of paths. For instance, in example 7.1, instead of generating the computation tree, we can generate a set of paths from initial states:
Although this method enables us to drive the UIs without the state rollback, if we directly evaluate the CTL formulas based on the set of sequences, the ability of testing the CTL formula is weaken. The reason is that we lost the relations between the branches. For instance, in the above example, the state 3 and state 4 should have same node. But with the restricted computation tree, although they still share the same node the state 2, they belong to the different paths. When carrying out the tests, we look at the path independently, so the state 3 and state 4 does not relate each other anymore. It results us only can test the CTL formulas with only one level quantifier. For the CTL formula like AG EF c.name=home, if we construct the test case with the restricted computation tree, it is impossible test such CTL formula. Because it requires that at each state, we need to examine the formula on the sub-paths from this state. With the restricted computation tree, there is no sub-paths defined.

To enable testing all kinds of CTL formulas, we reconstruct the computation tree based on the set of sequences which are generated for driving the real UIs. For each two sequences, we combine the elements with the same position in the sequence and the same state name in the UI model as a node in the tree. Recursively applying this method to all sequences results a computation tree. For instance, the sequences in above diagram can construct a computation tree which is shown at beginning of this section.

Generating the test cases

With the UI model, we can randomly generate a set of paths. The maximum number of states per path is specified before the generation. During generating a path, if encountering the state which already occurred before in this path, the generation of this path is complete immediately. This is because if a state is repeated in a path, the sequence of the states appear after this state can be represented by another path.

Example 7.2 For example, if there is a sequence of states like:
\[ s0 \rightarrow s1 \rightarrow s2 \rightarrow s3 \rightarrow s1 \rightarrow s4 \rightarrow s5 \rightarrow ... \]

It can separated into:
\[ s0 \rightarrow s1 \rightarrow s2 \rightarrow s3 \rightarrow s1 \]
\[ s0 \rightarrow s1 \rightarrow s4 \rightarrow s5 \rightarrow ... \]

Evaluate CTL formulas on the computation tree

In chapter 6, we describe How to use the SMV model check to evaluate the CTL formulas on the flattened UI model. Since we reconstruct the computation tree from a set of sequences, we can convert the computation tree into a flattened UI model. Then we can reuse SMV to evaluate the CTL formulas.
Example 7.3 Suppose the computation tree described in example 7.1 is reconstructed from the sequences that are used to drive the real UIs. For each node, we retrieve a UI attributes: words as:

0:words:6
2:words:2
0:words:6
3:words:4
4:words:3
2:words:5

By using SMV, we can reuse the CTL parser and the definitions of CTL that are already built-in SMV.

If we assign a unique identifier for each node, make each node as a state, each edge as a transition, we get a new UI model. Furthermore, since the UI attributes retrieved from the real UIs are directly bound to each node which means there is always two levels for each node, this new UI model actually is a flattened UI model. The unique identifier is given by using the node’s depth proceeding the original state name. Now the new flattened UI model is defined as:

- \( S = \{10, 11, 12, 22, 23, 24\} \)
- \( I = \{10\} \)
- \( R = \{(10, 11), (10, 12), (11, 12), (12, 13), (12, 14)\} \)
- \( C = \{10 \mapsto (0, \{(\text{words}, 6)\}), 11 \mapsto (1, \{(\text{words}, 2)\}), 12 \mapsto (2, \{(\text{words}, 1)\}), 23 \mapsto (3, \{(\text{words}, 4)\}), 24 \mapsto (4, \{(\text{words}, 3)\}), 22 \mapsto (2, \{(\text{words}, 5)\})\} \)

Now, we can convert the new flattened UI model which contains the UI attributes from the real UIs into the SMV program to evaluate the CTL formulas with SMV model checker.

7.1.2 Drive the real UIs and Retrieve the UI attributes

To drive the real UIs and retrieve the UI attributes, we should have ability to interact with the real UIs. Constructing a tool which can simulate users’ actions like clicking, moving mouse, typing is very complicated. In our framework, we render a open source tool called selenium to simulate users’ actions.

Selenium

Selenium is a tool which can automate web application testing. A selenium test defines the expected UI status and a sequence of actions which simulates the users’ actions. A UI status describes how the UI should look like, e.g, the element A is visible, the content of the link B should be “home”. When running the selenium test, the selenium engine can carry out actions and evaluate the expected UI status against the real web application.

A typical selenium system includes three parts: selenium core, selenium tests and selenium server. The selenium core is a javascript library which will be loaded into the browser when the selenium tests are launched and runs a web application inside an iframe HTML element. The selenium core can trigger actions on HTML elements like clicks, double clicks, drag, type in, etc.
The selenium tests can be written in the many programming languages like Java, Python, PHP, etc, and within different testing frameworks like JUnit, NUnit, unittest, etc. The selenium test defines the url of web application that will be tested, a sequence of actions and a set of assertions that describe the expected UI status. For example, within the JUnit test framework, we can open a web application by calling

```java
selenium.open("http://www.google.com/webhp?hl=en");
```

Typing in some texts in inputbox with id "q" can be done by

```java
selenium.type("q", "Seleniumtest")
```

we can assert title by

```java
assertEquals("Google", selenium.getTitle())
```

The selenium server can launch a browser and injects the selenium core library into it. It pushes the sequence of the users’ actions defined in the selenium tests into a queue and the selenium core will periodically query this queue and pick up the users’ actions, apply those actions to the real web application. When there is a request for evaluating the expected UI status, it also retrieves the UI attributes’ value from the real web application. The whole architecture of a typical selenium system looks like:

Another diagram which can help to understand typical selenium system
With above architecture, each selenium test can only match one instance of selenium server and selenium core. But the selenium server can run tests against the different environments like the different browsers, operating systems, etc. So the ideal architecture would be running one selenium test that can paralleled apply the test in the different environments. To do so, the selenium provides a master server called selenium Hub. The selenium hub is similar with the selenium server, but additionally, it keeps a table of the different selenium servers, and broadcasts the actions to those selenium servers. The new architecture looks like:

Now, if we want to run certain selenium test in different environments paralleled, we just need to setup one selenium hub, one selenium server for each environment and register the selenium
servers in selenium hub.

**Drive the real UIs and Retrieve the UI attributes with selenium**

Although the test cases that are generated from the UI model describe how the states of the UIs change, to transform the test cases into the selenium test cases, we still need the mapping between the transitions and the actual user actions and the relations between the UI elements’ id we used in the CTL formulas and the UI elements’ id in the real UIs, e.g., (state 2, state 3) is mapped to user action: clicking link A. Those extra information is supposed to be prepared by the UI engineers within a configuration file.

In our prototype, there is a module called TestLauncher which generates the test cases from the UI model and transforms the test cases into the selenium test cases with the configuration file.

At each state, we need the selenium to retrieve the UI attributes’ values from the real UIs. We summarize some UI attributes that can be retrieved by selenium below:

<table>
<thead>
<tr>
<th>UI attributes</th>
<th>selenium method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Height</td>
<td>getElementHeight</td>
</tr>
<tr>
<td>Left</td>
<td>getElementPositionLeft</td>
</tr>
<tr>
<td>Top</td>
<td>getElementPositionTop</td>
</tr>
<tr>
<td>Width</td>
<td>getElementWidth</td>
</tr>
<tr>
<td>Content</td>
<td>getText or getValue</td>
</tr>
<tr>
<td>Visible</td>
<td>isElementVisible</td>
</tr>
<tr>
<td>Amount of links</td>
<td>count number of the elements from the object array returned from getAllLinks method</td>
</tr>
<tr>
<td>Amount of inputboxes</td>
<td>count number of the elements from the object array returned from getAllFields method</td>
</tr>
<tr>
<td>Amount of buttons</td>
<td>count number of the elements from the object array returned from getAllButtons method</td>
</tr>
</tbody>
</table>
7.2 UI memory leaks testing

7.2.1 Memory leaks in web application

One of the critical measurements for the web application is the availability. It is important for the web application to provide 7/24 service since the users may access from all over the world via the internet. The web application becomes not available when it is crashed or no response. It can be caused by e.g., the programmatic problems, the out of memory errors, etc. There are many reasons that can result the out of memory errors. One of the main reasons is the memory leak. In this thesis, we show a way that how to test memory leaks by benefiting from our framework.

The memory leaks in the UI mostly exist in the browser. Some users’ actions may cause some memory leaks. They are difficult found immediately because the browser may be crashed after some time that the users interact with the web application. Although the browser memory usage can be viewed in operating system’s process manager like windows task manager, etc, such process manager can only provide information about how much memory is used by the browser process, it does not provide any useful information to help to allocate the potential parts of web application that leak the memory.

Before looking at how to test the memory leaks, we give a brief introduction that why there are memory leaks. Many web applications’ client side are written with the javascript. The javascript can perform some functions within the browser. The browser will allocate the CPU and memory to run the javascript program. The browser also interprets the HTML codes and provide the web page representation. Further, the browser builds a DOM system which is always consistent with the web page representation. By doing this, the javascript program can perform the actions on the web page by interacting with the relevant DOM system.

The javascript program can cause the memory leak by a circular reference. A circular reference is formed by two objects reference each other. In[], it indicates that “In a purely garbage collected system, a circular reference is not a problem: If neither of the objects involved is referenced by any other object, then both are garbage collected. In a reference counting system, however, neither of the objects can be destroyed, because the reference count never reaches zero. In a hybrid system, where both garbage collection and reference counting are being used, leaks occur because the system fails to identify a circular reference. In this case, neither the DOM object nor the JavaScript object is destroyed.”

Example 7.4 A simple example illustrates a circular reference can be:

```javascript
function JSObject ( domObject ) {
  this.domObject = domObject;
  this.domObject.JSObject = this;
}

var domObject = document.getElementById(‘domObject’); var jsObject = new JSObject(domObject);
```

Beside memory leak caused by the circular reference, there is another kinds of memory leak called Pseudo-Leaks. The different between the Pseudo-Leaks and the circular reference leak is that the Pseudo-Leaks only leaks memory while the web application is running in the browser, after the users close the browser, the memory will be release back. The Pseudo-Leaks may also causes the browser hanged if users use web application for a long period.

Example 7.5 A simple example illustrates a Pseudo-leaks can be:

```html
<html>
  <head>
    <script language="javascript">
```

45
function LeakMemory() {
// Do it a lot, look at Task Manager for memory response
var element = document.getElementById('test');
for (i = 0; i < 5000; i++)
{
    var newElement = document.createElement('Div');
    element.appendChild(newElement);
    element.removeChild(newElement);
}
}
</script>
</head>
<body>
<button onclick="LeakMemory()">Memory Leaking Insert</button>
<div id="test"></div>
</body>
</html>

In above example, we use a big loop to frequently add a new DOM element to the div with id ‘test’ and then immediately remove it by using the `removeChild` command; This is cause the pseudo-leaks particularly in the internet explorer since the `removeChild` command in the internet explorer just removes the DOM element from the web page visually, but the internet explorer still keeps a reference to that DOM element. When we run this program, the memory graph may look like:

To remedy this, one solution is using the `innerHTML=""` instead of the `removeChild`. So in the above program, we may adapt it into:

```
<html> ...

    function LeakMemory() {
        // Do it a lot, look at Task Manager for memory response
        var element = document.getElementById('test');
        for (i = 0; i < 5000; i++)
        {
            var newElement = document.createElement('Div');
            element.appendChild(newElement);
            element.innerHTML = "";
        }
    }

... </html>
```

Now, everytime, the memory will be freed immediately and the memory graph may looks like:
7.2.2 sIEve

sIEve is an open source tool that can detect memory leaks. It runs the web application inside, exposes all DOM elements status and also warns which DOM element leaks memory. Below is the screenshot of the sIEve main frame. It shows the memory usage and the amount of the DOM elements at the bottom while testers still can interact with the web application.

The sIEve also can show the details of the DOM elements. In the table shown below, the Orphan column shows if a DOM element causes Pseudo-Leaks. The Leaks column show if a DOM element causes circular reference leaks.

The sIEve still requires testers to manually interact with the web application. But the memory leak may becomes visible after a while or by certain action. So the completeness and the workloads becomes a problem for sIEve.
7.2.3 Our approach

From the UI model, we can randomly generate a path. We can apply this path to a real UI with selenium. Since both of the selenium and the sIEve are opensource tool, we create an adapter between those tools so that we can enable the selenium launches the sIEve and sIEve can run the selenium inside. Comparing with only using sIEve, the workloads are reduced by automation with selenium and the completeness can be improved by increasing the length of the path.
Chapter 8

Feasibility

In previous chapters, we describe the new approach. Applying the new approach requires the changes to the existing development lifecycle. The more changes required, the more difficult it is to apply the new approach. Furthermore, the benefits from the new approach are also an important factor that if the new approach is worth. We will discuss the feasibility of the new approach.

8.1 Current development lifecycle

The development lifecycle of the web based UI is as follow:

- Defining the UI specifications
  As indicated in chapter [], the UI engineers have to analyze the UI requirements, then define the UI specifications.

- The UI design
  At the UI design phase, the UI engineers give the UI designs which satisfy the UI specifications. The UI engineers normally first build the structure of all UIs then design the details of each UI. They visualize the design in pictures without any functional implementation.

- The UI design review
  After the UI design phase, the UI engineers review the UI design and try to find out the defects exist in the UI design which break the UI specifications.

- Implementation
  The software engineers implement the UI according the UI design.

- UI Testing
  After the UIs are implemented, the UI engineers may need to test the implementation against the UI design the UI specifications.

8.2 Working with our approach

Our framework tends to automate the works which are done by manually, so the benefits from our framework are mainly about the mistakes prevention. In this section, we describe the mistakes that can be made in the current development lifecycle, how our approach works and prevents those mistakes and the required changes to the current development lifecycle to apply our approach.

- The UI design breaks the UI specifications
  The UI engineers may not be able to remember all of the UI specifications. It results that UI design may contains the mistakes that break the UI specifications. Although those mistakes can be found at the UI design review phase, the current review procedure is carried out manually; it is error prone due to manual works and completeness.

  Instead of manual review, our approach represents all the UI specifications in CTL formulas,
uses UI models to represent the UI design. The manual review procedure is replaced by verifica-
tion, which is done automatically by a model checker. The model checker ensures the comple-
teness of the verification. Furthermore, the UI models are automatically generated after the UI engineer-
ners finish their UI design.
Applying our approach requires the UI engineers to do the UI design with our GUIBuilder which can generate UI models. The UI specifications should be defined in CTL formulas.

- The UI implementation breaks the UI specifications
  When software engineers implement the UIs according the UI designs, they may make mis-
takes and give a wrong implementation which breaks the UI specifications. At the UI testing phase, manual testing is obviously very time consuming. The same test cases may be repeated several times for different versions of the implementation. Complete re-testing can make a huge amount of work.
  Our approach automatically generates the test cases from UI models and applies them to the UI implementation.
  Instead of manual testing, the UI engineers have to learn how to make our automated testing tool running.

The diagram below shows the normal development lifecycle and how our approach is applied.

The phases with blue color are in the normal development lifecycle. The phases with red color are proposed in our approach.

The diagram shows the relations between our approach and the current development lifecycle. From the diagram, it is clear that the flow of the current development lifecycle is not changed by our approach. The UI design review is fully automated by the UI verification and the UI testing is replaced by the automated testing.

Above of all, our approach can be applied without changing the current development lifecycle. And the UI engineers can benefit from our approach for avoiding manually works, preventing the mistake and reducing the total amount of works.
Chapter 9

Case study

In this case study, we use wiki website from the computer science department at Utrecht university as an example. For the memory leak testing, we choose the google maps as our example, because at google maps, each map tile is a DOM element, and every action on the map will cause to delete and recreate all DOM elements. Frequently deleting and creating DOM elements can cause the memory leaks easily. We illustrate how our approach works that includes:

• Represent some UI specifications in the CTL formulas
• Model part of the wiki website
• Verify the UI specifications against the UI model of the wiki website
• Test the UI specifications against real wiki website
• Test the memory leaks in the google maps

9.1 wiki website UI specifications

Based on wiki website, we summarize some UI specifications.

• Each state should have at least one and only one of login link or logout link.
  It is important that users always can log in and log out. We know that at the login page, there of course does not have any login or logout link. so when we build the CTL formula, we should exclude login page. The statement is represented in the CTL formula as:

  \[ AG(c.name! = Login \land c.name! = \text{LoginPublications} \land c.name! = \text{LoginFAQ} \land c.name! = \text{LoginMasterHome} \land \neg c.name! = \text{LoginIntroduction} \rightarrow (\neg c.login.isVisible \land \neg c.logout.isVisible) \lor (c.login.isVisible \land c.logout.isVisible) \]

• Every page has a link back to the Centre’s page

  \[ AG EF(c.id = Center \lor c.id = \text{CenterLogin}) \]

• Whatever user is, publications page should be reachable within 3 clicks

  \[ AG(EBF 0..3 (c.id = \text{publications} \lor c.id = \text{publicationsLogin}) \]

• the user personal information should not be visible until user successful login

  \[ EF (c.id = \text{Login} \rightarrow A [\neg c.personalinfo.isVisible U c.logout.isVisible]) \]

• The department logo should always visible at center page

  \[ AG ((c.id = Center \lor c.id = \text{CenterLogin}) \rightarrow c.logo.isVisible) \]
• it is always possible that after publications is visited, it can be visited later again.

\[ EG \text{ (c.id = Publications) } \rightarrow \text{ EX (EF (c.id = Publications)))} \]

• it is always possible that at future user can visit Master program page

\[ AG \text{ EF(c.id = MasterHome} \lor \text{ c.id = MasterHomeLogin) } \]

9.2 wiki website UI model

The current wiki website contains many web pages. In this case study, we model parts of the wiki website that we think the model is enough for illustrating how our approach works. The UI model includes Center page, Publications page, FAQ page, Login page, MasterHome page and Introduction page. The UI model is established based on page level.

The structural UI model without UI attributes are represented by:

The structural UI model without the UI details are defined as:

- \( S = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\} \)
- \( I = \{0\} \)
- \( R = \{(0, 0), (0, 1), (0, 2), (0, 7), (0, 3), (1, 1), (1, 0), (1, 2), (1, 11), (1, 7), (2, 2), (2, 0), (2, 1), (2, 12), (2, 7), (7, 7), (7, 0), (7, 8), (7, 13), (8, 8), (8, 0), (8, 7), (8, 14), (3, 3), (3, 4), (11, 11), (11, 5), (12, 12), (12, 6), (13, 13), (13, 9), (14, 14), (14, 10), (4, 4), (4, 5), (4, 6), (4, 9), (4, 0), (5, 5), (5, 4), (5, 6), (5, 1), (5, 9), (6, 6), (6, 4), (6, 5), (6, 2), (6, 9), (9, 9), (9, 4), (9, 10), (9, 7), (10, 10), (10, 4), (10, 9), (10, 8)\} \)
- \( C = \{0 \leftrightarrow (id : Center), 1 \leftrightarrow (id : Publications), 2 \leftrightarrow (id : FAQ), 3 \leftrightarrow (id : Login), 4 \leftrightarrow (id : CenterLogin), 5 \leftrightarrow (id : PublicationsLogin), 6 \leftrightarrow (id : FAQLogin), 7 \leftrightarrow (id : MasterHome), 8 \leftrightarrow (id : Introduction), 9 \leftrightarrow (id : MasterHomeLogin), 10 \leftrightarrow (id : IntroductionLogin), 11 \leftrightarrow (id : LoginPublications), 12 \leftrightarrow (id : LoginFAQ), 13 \leftrightarrow (id : LoginMasterHome), 14 \leftrightarrow (id : LoginIntroduction)\} \)

In the real implementation, if users reach login page from center page, after successful login, they must go to center page with login status. To catch this information in the UI model, we duplicate login page into 5 different states, each state matches a single page.

The complete UI model can be generated from GUI builder. The screenshot for each state in GUI builder can be found at the appendix 11.2
9.3 Verification

With the UI model generated by the GUI builder and the UI specifications represented in the CTL formulas, the SMVConverter module generates the scripts which accepted by the SMV model checker.

Below is the output from SMV model checker:

```plaintext
To illustrate a example which the CTL formula is failed, we change the formula

\[ AG((c.name! = Login) \land c.name! = LoginPublications \land c.name! = LoginFAQ \land c.name! = LoginMasterHome \land c.name! = LoginIntroduction) \rightarrow (c.loginisVisible \land \neg c.logout.isVisible) \lor (\neg c.login.isVisible \land c.logout.isVisible) \]

into

\[ AG(c.login.isVisible \land \neg c.logout.isVisible) \lor (\neg c.login.isVisible \land c.logout.isVisible) \]

This CTL formula is wrong because at the login page, there is not login link and logout link.

Now the result from SMV model checker looks like:
```
The SMV model checker detects this CTL formula is failed and also gives a counter example in the output.

9.4 Testing

Due to the limitation of UI testing, in this case study, we only test the formula:

$$\text{AG}(\text{c.name!} = \text{Login} \land \text{c.name!} = \text{LoginPublications} \land \text{c.name!} = \text{LoginFAQ} \land \text{c.name!} = \text{LoginMasterHome}$$

$$\land \text{c.name!} = \text{LoginIntroduction} \rightarrow (\text{c.login.isVisible} \land \neg \text{c.logout.isVisible}) \lor (\neg \text{c.login.isVisible} \land \text{c.logout.isVisible})$$

Below is the output from TestLauncher:

9.5 Memory leak detection

The Google map allows users to view the map and navigate the map by panning left, panning right, panning up, panning down, zooming out and zooming in. To test memory leaks, we want TestLauncher can randomly generate the test cases from the UI model and use those test cases to stress
the Google Map application for a long time and monitor the memory usage and find possible memory leaks by the sIEve.

The structural UI model of the Google Map looks like:

```
S = {0, 1, 2, 3, 4, 5}
I = {0}
R = {(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (2, 0), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (3, 0), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (4, 0), (4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (5, 0), (5, 1), (5, 2), (5, 3), (5, 4), (5, 5)}
C = {0 \mapsto \text{id: PanUp}, 1 \mapsto \text{id: PanDown}, 2 \mapsto \text{id: PanLeft}, 3 \mapsto \text{id: PanRight}, 4 \mapsto \text{id: ZoomIn}, 5 \mapsto \text{id: ZoomOut}}
```

With this UI model, we randomly navigate the Google Map in the sIEve 1 hour. Below is the result from the sIEve:

The memory usage definitely increased from 31412 KB to 113564 KB. And sIEve also shows which DOM element leaks the memory:
So the DOM element marks with orphan means it cause the Pseudo-Leaks. And from the URL of that DOM element, we know which part leaks memory.

After we know which part of the web application may leaks the memory, we can restrict the UI model by taking the subset to enlarge the memory leak. By doing this, we can see the memory leaks easier when we test the memory leaks again.

The restricted UI model without the UI details are defines as:

- \( S = \{0, 1, 2, 3, 4\} \)
- \( I = \{4\} \)
- \( R = \{(0, 0), (0, 3), (0, 4), (1, 0), (1, 1), (1, 2), (1, 4), (2, 0), (2, 2), (2, 4), (3, 1), (3, 3), (3, 4), (4, 0), (4, 4)\} \)
- \( C = \{0 \mapsto (id : PanUp), 1 \mapsto (id : PanDown), 2 \mapsto (id : PanLeft), 3 \mapsto (id : PanRight), 4 \mapsto (id : ZoomIn)\} \)
The test case generated from this UI model results the memory usage increased to 112136 KB in 40 minutes. The memory usage diagram from sIEve looks like:
Chapter 10

Conclusion

In this thesis, we investigate the current development lifecycle of the web based UI and find out that there are several phases are completed manually which is error prone and time consumed. Based on our investigation, we tend to automate the web based UI design by proposing an approach which represent the UI design in the UI model, automatically verifying the UI specifications with the UI model, generated the test cases based on the UI model and automatically applied the tests against the real implementation.

Besides showing the strength of our approach, we also show the feasibility of our approach that there are not many efforts required to adapt the current development lifecycle to apply our approach. The UI engineers can just remain their original working flow but facility from our approach to automate some phases.

We summarize the pros and cons of our approach:

- Pros
  - The design review phase are completely automated and the model checker can ensure the completeness.
  - The UI testing phase are completely automated.
  - Our approach does not impact the current development lifecycle a lot.

- Cons
  - Our approach requires the UI engineers have some CTL knowledge.
  - The GUI builder we implement for demonstrating our approach currently only can give a rough graphical design.

Although the tool we currently implemented has some rough edges, it approves our approach is feasible and quite useful for the web based UI design. It is very flexible and has lots of potentials that can be further extended like using different techniques to represent the UI specifications to take more kinds of the UI specifications into analysis, invite more effective test algorithm, make editing CTL, memory monitor more graphical, etc.
Chapter 11

Appendix

11.1 Appendix 1

The UI model for example 4.4 is:

- $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\}$
- $I = \{0\}$
- $R = \{(0, 1), (0, 11), (1, 0), (1, 2), (1, 4), (2, 0), (2, 3), (2, 4), (3, 0), (3, 4), (3, 5), (4, 0), (5, 0), (5, 6), (6, 7), (6, 10), (6, 9), (7, 6), (7, 10), (7, 8), (8, 7), (8, 5), (9, 8), (9, 10), (9, 6), (10, 6), (11, 12), (11, 13), (11, 14), (12, 11), (12, 13), (12, 14), (13, 11), (13, 12), (13, 14), (14, 11), (14, 12), (14, 13)\}$
- $C = \{0 \rightarrow (id : \text{home}), 1 \rightarrow (id : \text{login}), 2 \rightarrow (id : \text{login}), 3 \rightarrow (id : \text{login}), 4 \rightarrow (id : \text{error}), 5 \rightarrow (id : \text{mailboxIndex}), 6 \rightarrow (id : \text{mailboxNewMail}), 7 \rightarrow (id : \text{mailboxNewMailBodyEmpty}), 8 \rightarrow (id : \text{mailboxNewMailReceiverBodyFilled}), 9 \rightarrow (id : \text{mailboxNewMailReceiverEmpty}), 10 \rightarrow (id : \text{error}), 11 \rightarrow (id : \text{privacyCenter}), 12 \rightarrow (id : \text{privacyOverview}), 13 \rightarrow (id : \text{privacyPolicy}), 14 \rightarrow (id : \text{privacyFAQ})\}$

To make above example more clear about which action causes which state, we explicitly give the mapping between transitions and actions below:
<table>
<thead>
<tr>
<th>Transition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,1)</td>
<td>click login link</td>
</tr>
<tr>
<td>(1,0)</td>
<td>click home link</td>
</tr>
<tr>
<td>(1,2)</td>
<td>type in username</td>
</tr>
<tr>
<td>(1,4)</td>
<td>click submit button</td>
</tr>
<tr>
<td>(2,3)</td>
<td>type in password</td>
</tr>
<tr>
<td>(2,4)</td>
<td>click submit button</td>
</tr>
<tr>
<td>(2,0)</td>
<td>click home link</td>
</tr>
<tr>
<td>(3,0)</td>
<td>click home link</td>
</tr>
<tr>
<td>(3,4)</td>
<td>click submit button</td>
</tr>
<tr>
<td>(3,5)</td>
<td>click submit button</td>
</tr>
<tr>
<td>(4,0)</td>
<td>click home link</td>
</tr>
<tr>
<td>(5,0)</td>
<td>click home link</td>
</tr>
<tr>
<td>(5,6)</td>
<td>click composeMail link</td>
</tr>
<tr>
<td>(6,7)</td>
<td>type in subject</td>
</tr>
<tr>
<td>(6,10)</td>
<td>click send button</td>
</tr>
<tr>
<td>(6,9)</td>
<td>type in mailBody</td>
</tr>
<tr>
<td>(7,6)</td>
<td>type in subject</td>
</tr>
<tr>
<td>(7,10)</td>
<td>click send button</td>
</tr>
<tr>
<td>(7,8)</td>
<td>type in mailBody</td>
</tr>
<tr>
<td>(8,7)</td>
<td>type in mailBody</td>
</tr>
<tr>
<td>(8,5)</td>
<td>click send button</td>
</tr>
<tr>
<td>(9,8)</td>
<td>type in subject</td>
</tr>
<tr>
<td>(9,10)</td>
<td>click send button</td>
</tr>
<tr>
<td>(9,6)</td>
<td>type in subject</td>
</tr>
<tr>
<td>(10,6)</td>
<td>click composeMail link</td>
</tr>
<tr>
<td>(0,11)</td>
<td>click privacy link</td>
</tr>
<tr>
<td>(11,12)</td>
<td>click privacyOverview link</td>
</tr>
<tr>
<td>(11,13)</td>
<td>click privacyPolicy link</td>
</tr>
<tr>
<td>(11,14)</td>
<td>click privacyFAQ link</td>
</tr>
<tr>
<td>(12,11)</td>
<td>click privacyCenter link</td>
</tr>
<tr>
<td>(12,13)</td>
<td>click privacyPolicy link</td>
</tr>
<tr>
<td>(12,14)</td>
<td>click privacyFAQ link</td>
</tr>
<tr>
<td>(13,11)</td>
<td>click privacyCenter link</td>
</tr>
<tr>
<td>(13,12)</td>
<td>click privacyOverview link</td>
</tr>
<tr>
<td>(13,14)</td>
<td>click privacyFAQ link</td>
</tr>
<tr>
<td>(14,11)</td>
<td>click privacyCenter link</td>
</tr>
<tr>
<td>(14,12)</td>
<td>click privacyOverview link</td>
</tr>
<tr>
<td>(14,13)</td>
<td>click privacyPolicy link</td>
</tr>
</tbody>
</table>
11.2 Appendix 2

Below is the screenshots of GUIBuilder for the example described in chapter 9.

The screenshot for designing **Center** page is

![Center page screenshot](image)

The screenshot for designing **FAQ** page is:

![FAQ page screenshot](image)
The screenshot for designing **Publication** page is

The screenshot for designing **Login** page is
The screenshot for designing **Master home page** is

![Master Program Software Technology](image)

The Master Program Software Technology: It covers techniques that are applied in the production of software. We start from sound theoretical foundations, with a strong focus on real applicability. The program includes courses on programming methodology, programming languages, programming tools, software architectures, component based programming, specification formalism and verification techniques.

The Master Program is internationally oriented: the program is open to foreign students, and courses are taught in English. Students have the opportunity to follow courses and do projects at foreign universities and institutes. The program caters for students with a desire to obtain a Ph.D. position or a research position in a company, and for students who are interested in jobs as software designer in industry. The program results in a degree Master of Science in Computer Science. The program is offered by the Center for Software Technology of the Institute of Information and Computing Sciences at Delft University.

**Outline**
- Introduction, our areas of study, and our curriculum,
- career profiles,
- supervision

The screenshot for designing **Introduction page** is

![Introduction](image)

The Software Technology for program studies of techniques that may be needed in the production of actual software, we do so for starting from sound theoretical foundations, with a strong focus on real applicability. It includes programming methodology, programming formalisms (languages), programming tools, software architectures, component based programming, specification formalism and verification techniques (constructive proofs, theorem proving).
After user login, for each page, at right top, the instead of login link, the user name and logout is shown. Below is the screenshot for web home page with user login:
Bibliography

[18] Scott D. Wood Tom Brinck, Darren Gergle. Usability for the web: Designing web sites that work.