Intelligent agent based automatic operating model

Yin, Zun Liang

2005

Yin, Z. L. (2005). Intelligent agent based automatic operating model. Master' s thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/3901>

<https://doi.org/10.32657/10356/3901>

Nanyang Technological University

Downloaded on 26 Apr 2023 19:41:30 SGT

An intelligent agent based automatic operating model

Yin Zun Liang

School of Electrical & Electronic Engineering

A thesis submitted to the Nanyang Technological University
in fulfillment of the requirement for the degree of
Master of Engineering

2005

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research and has not been submitted for a higher degree to any other University or Institution

Date

Yin Zun Liang

ACKNOWLEDGEMENTS

I would like to express my gratitude towards my supervisor, Dr. Miao Yuan, for his invaluable advice and guidance throughout the course of the project. Faith and confidence entrusted by Dr. Miao has facilitated the progress of the research and is deeply appreciated.

I would also thank Dr Huang Guang Bin for his helpful advice.

SUMMARY

Human activities have largely converged to desktop computers. We perform various tasks through computer interfaces (CI), such as learning (e-learning), shopping (e-commerce), etc. Today's major operating systems provide a graphical user interface (GUI). These applications have become the dominant ones used by end-users. Many of the tasks or operations over the CI are repeated. It is highly desired that these operations could be performed automatically.

We identify two types of GUI applications to be automated. The first is applications which have been released as products and are operated by end-users to achieve tasks. The second type is applications which are being tested to identify possible bugs.

GUIs are used by users or testers to interact with the applications. Since logics are not explicitly presented on the GUIs, the automation relies on actionable knowledge of the users and testers when they operate the GUI applications.

At present, there are no effective techniques for GUI application automation. In practice, “capture/replay” is the main approach used for the automation of GUI applications. Using this approach, a tool captures all the interactions with the GUI application and stores these actions into script files. These actions include keystrokes, mouse button clicks, etc. At a later time when the tool replays the recorded scripts, it initiates the same action that the original user performed.

Using the “capture/replay” approach to automate GUI application is limited to replaying of exactly what has recorded. When automating a product application, it lacks the ability to take different actions according to the execution context of the running GUI application. When automating an application under test, it lacks the ability to adapt the change of the GUI application.

In this thesis we first discuss a general automatic model for intelligent agent based operations. Since applications under test and applications released as products are operated for different purposes, two elaborated model are derived from the general model: intelligent agent based operating Model for Product Applications (MPA) and intelligent agent based operating Models for Regression Testing (MRT).

The MPA is used to help the user to perform tasks and the MRT is used to help the tester to test GUI applications regressively.

In MPA, the knowledge is represented as rules. In this model, five agents are used to do the GUI automation: user interface agent, playing agent, monitoring agent, reasoning agent and input data reading agent. At capture phase, a capturing application is used to record user's actions and convert these actions into rules. At automation phase, the user assigns tasks through the user interface agent. The playing agent is used to regenerate mouse and keyboard events and send them to the underlying operating system. The monitoring agent is used to monitor the GUI states of an application. The input data reading agent is used to obtain input data and provide them to the reasoning agent. The reasoning agent is used to do the reasoning work in response to the change of states of the execution environment perceived by the monitoring or input data reading agent. According to execution context of a running GUI application, these agents work together to drive a GUI application to perform a specific task automatically.

In the MRT, the knowledge is represented as an actionable knowledge model, which is able to adaptive to the change of GUI applications. Seven agents are identified in the model: the user interface agent, the capturing agent, the playing agent, the remote screen capturing agent, the remote controlling agent, the KB managing agent and the mediating agent. The user interface agent is used to accept instructions from the tester and displays information generated by the mediating agent. The capturing agent is used to capture the tester's actions when the tester operates the GUI application under test. The playing agent is used to perform the GUI actions of a test case. The remote screen capturing agent is used to capture the screen image of a remote GUI application and send them to the remote controlling agent. The remote controlling agent is used by the tester to display the remote screens and to control the remote GUIs. The KB managing agent is used to convert, search and reconstruct test cases. The mediating agent plays the role of mediator in the system. It processes the tester's command, relay messages between other agents, and keep log information. These agents help the tester to create, execute and repair test cases.

Two prototypes were developed to verify the MPA and the MRT.

TABLE OF CONTENTS

Acknowledgements	i
Summary	ii
List of Figures	vi
Chapter 1 Introduction	1
1.1 Motivation	1
1.2 Objectives	3
1.3 Major contribution of the Thesis	3
1.4 Organization of the Thesis	3
Chapter 2 Background.....	5
2.1 GUI applications.....	5
2.2 Expert system shell.....	8
2.3 Software agents	10
2.4 Capture/replay tools	15
2.5 Summary.....	21
Chapter 3 Intelligent Agent Based Automatic Operating Model.....	22
3.1 Definition of GUI model.....	23
3.2 Intelligent agent based automatic operating model	25
3.3 Two types of activities automation	26
3.4 Summary.....	30
Chapter 4 Automatic Operating Model for Product Applications	31
4.1 Overview	31
4.2 System architecture	41
4.3 Capturing application	42
4.4 Rules representation	48
4.5 Agents definitions.....	51
4.6 Agents interactions	67
4.7 Summary.....	70
Chapter 5 Automatic Operating Model for GUI Regression Testing.....	72
5.1 Overview	72
5.2 Test case conversion algorithm.....	81

5.3	Test case reconstruction algorithm	90
5.4	Test case repair related algorithm	92
5.5	Architecture of the proposed GUI regression testing model.....	95
5.6	Agent interactions.....	96
5.7	Summary.....	98
Chapter 6	Prototype Implementation	99
6.1	Prototype implementation of GUI automation model.....	99
6.2	Prototype implementation of GUI regression testing model	108
6.3	Summary.....	118
Chapter 7	Conclusion and Recommendations.....	119
7.1	Conclusion	119
7.2	Recommendations for further research.....	121
Author's Publications		123
Bibliography		124

LIST OF FIGURES

Figure 2-1 Sample window GUI	6
Figure 2-2 Standalone GUI application architecture	7
Figure 2-3 Client Server GUI application architecture.....	7
Figure 2-4 Distributed GUI application architecture	7
Figure 2-5 Architecture of expert system.....	8
Figure 2-6 Sample ACL message.....	13
Figure 2-7 Reference architecture of a FIPA Agent Platform.....	15
Figure 2-8 JADE Agent Platform over Several Containers	15
Figure 3-1 General intelligent agent based automatic operating model.....	26
Figure 4-1 Actions in a script file.....	36
Figure 4-2 Screen shot of Microsoft Outlook	36
Figure 4-3 Sample action flow of reading email.....	37
Figure 4-4 example of repeated work.....	39
Figure 4-5 Exception process	41
Figure 4-6 Agent Based GUI automation model architecture.....	41
Figure 4-7 Components in the capturing application.....	44
Figure 4-8 Example GUI application used for GUI automation	45
Figure 4-9 GUI automation agents' interaction scenario for expected states	69
Figure 4-10 GUI automation agents' interaction scenario for unexpected states	70
Figure 5-1 AKG example	77
Figure 5-2 AKG example for test case repair automatically	79
Figure 5-3 AKG example for test case repair manually	80
Figure 5-4 Example GUI for regression testing.....	86
Figure 5-5 AKG for test case T1	87
Figure 5-6 GUI example for add existing staff.....	88
Figure 5-7 AKG for test case T1 and T2.....	89
Figure 5-8 Architecture of multi-agent based GUI regression testing model.....	95
Figure 6-1 Screenshot of capturing application	100
Figure 6-2 Screenshot of agent interaction simulation application	102
Figure 6-3 A scenario of agents' deployment.....	109
Figure 6-4 Screenshot of JADE remote monitoring agent	111

Figure 6-5 Screenshot of remote screen captured by the remote controlling agent (left)
..... 111

GLOSSARY

ACC	Agent Communication Channel
ACL	Agent Communication Language
AI	Artificial Intelligence
AK	Action Knowledge
AKG	Action Knowledgeable Graph
AMS	Agent Management System
AUT	Application Under Test
BDI	Belief-Desire-Intention
CI	Computer Interface
DF	Directory Facilitator
FIPA	Foundation for Intelligent Physical Agent
G-CFG	Control-flow graph
GUI	Graphical User Interface
JVM	Java Virtual Machine
MPA	Intelligent Agent Based Automatic Operating Model for Product Automation
MRT	Intelligent Agent Based Automatic Operating Model for Regression Testing
PA	products applications

RMI	Remote Method Invocation
SGL	Sub-Goals-List
SL	Semantic Language
TSL	Mercury Interactive's Test Script Language

Chapter 1 Introduction

1.1 Motivation

This research is inspired by the following motivation:

Human activities have largely converged to desktops. We perform various tasks through computer interfaces (CI), such as learning (e-learning), shopping (e-commerce), etc. Today's major operating systems provide a graphical user interface. Graphical User Interface (GUI) has become the most important way of interacting with computer systems. Most of the GUI applications are event driven and interactive. The user operates the GUI application to achieve a task using a mouse and a keyboard. The GUI application reacts based on both the actions the user has performed and the underlying business logic. The user takes further actions based on the states (screen outputs) of the GUI application. This makes GUI applications intuitive and easy to use.

At the same time, a GUI application loses the ability to redirect its inputs and outputs as compared to a traditional application. This makes it is extremely hard if at all possible to run an interactive GUI application without user's interference. Since operating a GUI application manually to perform repeatable tasks each time is a waste of time, GUI application automation is highly desired.

At present, there are no effective techniques for GUI application automation. The most popular approach to automate GUI application is the "capture/replay" approach and this approach is mainly used to automate GUI application regression testing [1] [2]. So far, researches on GUI applications testing are related to GUI test case generation, selection [3] [4] [5] [6] [7]. Regression testing of GUIs is still remained a largely unexplored area [1].

The characteristics of the GUI applications under test are different from that of the released product GUI applications. One difference is that the GUI applications under test are changed more often than the released GUI applications. Another difference is that GUI automation aims to perform a task automatically on behalf of the user. It is

desired to be able to play different actions based on the states of a running GUI application at some execution point. However, GUI application regression testing aims at finding bugs. When a GUI application is modified, it checks whether the modified GUI application still conforms to the specification.

When a “capture/replay tool” is used to automate a GUI application or test a GUI application regressively, the user or tester uses a capture tool to record what he has performed on the GUI application into script files. At a later time, the “capture/replay” tool could replay these recorded script files to operate the GUI application automatically. The drawback of the “capture/replay” approach is as follows:

- In GUI application automation, the replay tool can only replay what exactly the user has performed. It lacks the ability to play different actions based on the execution context of a running GUI application. To automate GUI applications, the automation tool needs the ability to act according to different normal and abnormal GUI states when a GUI application is running.
- In GUI regression testing, if the GUI is modified at a later time, then some of the recorded test cases will become invalid. Re-recording or manually modifying the script files is needed. Manually modification to the script files needs professional users and recording all the script files totally from the beginning is very time consuming.

In this thesis, we identify two types of GUI applications to be automated. One type is applications which have been released as products and are operated by end-users to achieve tasks. The other type is applications which are being tested and operated by testers to identify bugs. To automate the two types of GUI applications, we first discuss the general agent-based GUI operating model. Then according to the different purposes of operating the two types of GUI applications, based on the general model, we propose two elaborated models: one for automation of product GUI applications and the other one for GUI regression testing.

1.2 Objectives

The objective of this thesis concerns the automation of GUI applications. It presents the intelligent agent based automatic operating model which can be applied to operate GUI applications automatically.

1.3 Major contribution of the Thesis

This thesis documents several key contributions made to the field of GUI application automation as well as GUI application regression testing.

- 1) We define an agent based operating model for the automation of product applications. In this thesis we demonstrate its usefulness in helping the user to automatically perform tasks using GUI applications.
- 2) We define an agent based operating model for regression testing of GUI applications. In this thesis we demonstrate its usefulness in helping the tester to test GUI applications regressively.
- 3) We use rules to represent the user actions and execution contexts of a GUI application. It enables different actions are taken based on a GUI application's execution context.
- 4) We propose the actionable knowledge model based presentation of test cases. It enables easy repair of test cases.

1.4 Organization of the Thesis

This thesis is organized as follows:

Chapter 2 presents an overview of GUI applications, agent technology, and expert system technology.

Chapter 3 discusses the general intelligent agent based automatic operating model.

Chapter 4 presents the proposed elaborated automation model for product GUI applications.

Chapter 5 presents the proposed elaborated automation model for regression testing.

Chapter 6 presents the prototype implementations of the GUI automatic operating model and the GUI application regression testing model.

Chapter 7 concludes the thesis and discusses the future work.

Chapter 2 Background

The purpose of this chapter is to provide the background of GUI applications, software agents, expert systems and “capture/replay” tools.

2.1 GUI applications

2.1.1 Characteristics of GUI applications

An important feature of a GUI application is the GUI. A GUI is composed of user interface elements, such as windows, dialog box, menus, buttons, icons, list box, text box, etc.

A GUI application is composed of windows. A window is composed of user interface elements, such as dialog boxes, menus, buttons, icons, list boxes, text boxes, etc. A sample window GUI is shown in Figure 2-1. Each GUI object has many properties, such as “class”, “label”, “width”, “height”, “handle”, “enabled”, etc. and each element can be uniquely distinguished by part or all of its properties.

At present most of the GUI applications run on a windowing system. Multiple GUI applications share the same computer’s graphical display presentation resources of a windowing system. A windowing system is an event driven system. Each hardware and user action causes one or more messages to be generated. Windows events are communicated via messages. For example, if a key is pressed, then WM_KEYDOWN, WM_KEYUP, and WM_CHAR message are generated. These messages can be captured and simulated. The output to the screen can also be captured and analyzed. A number of windows from one or more GUI applications may be displayed on the screen at the same time. Only one window has the focus. Hardware events are only sent to the window with focus. “Capture/replay” tools, such as Mercury Interactive’s WinRunner [8] and IBM’s Rational Robot [9], are able to capture the mouse and keyboard events.

Figure 2-1 Sample window GUI

2.1.2 Category of GUI applications

We can categorize GUI applications in many different ways. In this thesis we categorize the GUI applications into the following categories:

- Monolithic applications.
- Client/Server applications.
- Distributed applications.

Monolithic applications run on the user's PC, read and write a local disk for files or databases. Its architect is shown in Figure 2-2. An example would be Microsoft Note Pad.

Figure 2-2 Standalone GUI application architecture

Client/Server applications run on the users' PC, reading and writing a shared disk of files or database server. Its architecture is shown in Figure 2-3. Most of the business process applications are of this kind.

Figure 2-3 Client Server GUI application architecture

Distributed applications run on several users' PCs, and communicate with each other. The architecture is shown in Figure 2-4. An example would be chat applications.

Figure 2-4 Distributed GUI application architecture

2.2 Expert system shell

2.2.1 What is an expert system and what is expert system shell

Expert systems are computer programs that employ human knowledge to solve problems, which ordinarily require human intelligence [10]. Expert system is one of a group of disciplines in Artificial Intelligence (AI).

Every expert system consists of two principal parts: the knowledge base and the reasoning (inference) engine. The architecture of expert systems is shown in Figure 2-5.

Figure 2-5 Architecture of expert system

An expert system shell is a general-purpose framework that contains no specific domain knowledge. It provides the schema to allow domain specific knowledge to be filled to construct an actual expert system.

The user interface is used for the user to interact with the expert system shell.

The inference engine controls rule matching, rule selection and rule firing.

The knowledge is the “if ...then” rules and problem instances called facts.

2.2.2 Method of inference

The method of inference of an expert system is a very important aspect if we want the best results from the system.

There are two primary methods for manipulating the knowledge that represents the rules: forward chaining and backward chaining.

Forward chaining, which is also called Data-Driven reasoning, takes facts of the problem, applies rules and moves toward some conclusion. Backward chaining, which is also called Goal-Driven reasoning, starts from a goal state back to the original state. It finds the rules that can produce the goal and chains backward through sub-goals to the given facts.

2.2.3 Knowledge representation

Facts and rules make up the knowledge base of a rule-based expert system's. A fact is a construct that defines a piece of information that is known to be true. A rule is an **if/then** statement that defines the set of facts of the “if” part that must be true before a set of actions of the “then” part can be executed. The format of a rule is shown as below:

If *condition 1, condition 2, ..., condition m*

Then *action 1, action 2, ..., action n*

2.2.4 Advantage of using expert system shell

Rule-based expert systems are extremely powerful because actions themselves can assert new facts. When this happens additional rules apply and their actions are executed.

There are significant advantages when building expert systems by using expert system shells. A system can be built to perform a unique task by entering into a shell all the necessary knowledge about a task domain.

In the expert system approach, the rule engine separates critical application rule from the rest of the application code. This makes the application user customize applications at runtime with no programmer involvement.

In an expert system approach the sequence of the rules is usually unimportant. The inference engine decides which rule to apply at any moment. If a rule's “if” part is

satisfied then the rule will be fired. Sequential discipline needed in conventional programming does not exist any more. The approach for solving the problem does not necessarily have to be known at the beginning.

2.3 Software agents

2.3.1 What is an agent

Agent technology is an area where formalization of principles and its definitions are still maturing [11]. Because of this general lack of formalization, it is difficult to define precisely what an agent is. In [11] the author gives a definition of an agent: “referring to a component of software and/or hardware which is capable of acting exactly in order to accomplish tasks on behalf of its user”.

2.3.2 What is a software agent

A Software Agent is a computer program, which performs tasks on behalf of another entity, possibly over an extended period of time, without direct supervision or control [12].

Here the other entity could be a human, a traditional, "legacy" computer program, a robot, or another software agent. A software agent is similar to a robot, but operates in cyberspace, on a computer network.

The features of a software agent are autonomous or at least semi-autonomous. It may or may not have a user interface. When performing tasks in pursuit of a goal, there could be no direct supervision or direct control. A software agent may interact with another agent to obtain guidance or output results.

2.3.3 What is an intelligent software agent

An intelligent software agent is a software agent that uses Artificial Intelligence (AI) in the pursuit of the goals of its clients [13].

Artificial Intelligence is the imitation of human intelligence. Intelligent agents need to work together on a user-specified problem when told to do so and must be able to do this successfully in a dynamic environment

2.3.4 Agent characteristics

Different agents have different properties. Some of the agents have some of the following properties [14]:

- Autonomous – an agent should be able to execute without human interaction, although intermittent interaction may be required.
- Social/communicative – an agent should have a high level of communication with other agents.
- Reactive/response – an agent should be able to perceive its environment and react to changes to it.
- Proactive – proactive agents do not just react to their environment but can take active steps to change that environment according to their own desires.
- Adaptive – adaptive agents have the ability to adjust their behavior overtime in response to internal knowledge or changes in the environment around them.
- Goal-oriented/intention – these agents have an internal state that remains consistent over time.
- Persistent/continuous – persistent agents have an internal state that remains consistent over time.
- Mobility – mobile agents can proactively decide to migrate to a different machine or network while maintaining persistence.
- Intelligence – agents with the ability to reason, learn and adapt over time.
- Honesty – agents that believe in the truthful nature of the information they pass on.

Agents can help easy user tasks and adapt to user requirements and are likely to become the next engineering paradigm for system development. An increasingly popular

programming paradigm is that of agent-oriented programming [15]. A number of methodologies have been reported to address agent-oriented software engineering [14].

2.3.5 Agent ontologies

Ontologies represent a world view in a particular domain which consists of concepts, definitions and concept-relationships. Ontologies provide agents a powerful domain of discourse. Before communication, agents can agree to use a particular ontology, which defines some aspect of the world. Using such a shared ontology, agent can then discuss a topic with complete confidence that other agents have the same understanding of the items discussed.

2.3.6 Agent communication

An Agent Communication Language (ACL) provides agents with a means to exchange information and knowledge [16] [17].

The ACL itself defines the types of messages (and their meaning) that agents may exchange. Agents though, do not just engage in single message exchanges but they have conversations, *i.e.* task-oriented, shared sequences of messages that they follow, such as a negotiation or an auction. At the same time, some higher-level conceptualization of the agent's strategies and behaviors drives the agent's communicative (and non-communicative) behavior. [16]

The FIPA (Foundation for Intelligent Physical Agent) Agent Communication Language (FIPA ACL) is based on speech act theory: messages are actions, or communicative acts, as they are intended to perform some action by virtue of being sent.

The Semantic Language (SL) is the formal language used to define FIPA ACL's semantics. In FIPA ACL, the semantics of each communicative act are specified as sets of SL formulae that describe the act's feasibility preconditions and its rational effect. A full description of SL can be found in [18].

Below are some examples of communicative acts

Inform: inform the recipient about a proposition that the sender believes true

Query-if: ask the recipient if a given proposition is true or not.

Request: request the recipient to execute a given action.

An ACL messages example is shown in Figure 2-6. In this example agent A asks agent B to open a file. The content is in VB format.

```
(request
:sender agentA
:receiver agentB
:content "open \"a.txt\" for input"
:language vb )
```

Figure 2-6 Sample ACL message

2.3.7 The BDI model

Agents are often described using logic [19], [20]. Concepts that are commonly incorporated in such logics are for instance, knowledge, beliefs, desires, intentions, commitments, goals and plans [21]. A BDI approach is one of the main approaches to reason with rational agents have emerged those last ten years: BDI approaches [22] [23] [24] [25].

The logical foundations of BDI systems are based on the philosophical concepts of intentions, plans, and practical reasoning developed by Bratman [26]. The BDI model has proven to be a good one for modeling intelligent behavior in rapidly changing worlds [27]. The BDI model is a popular model for intelligent agents. The central concepts in the BDI model are:

- Beliefs: Information about the environment.
- Desires: Objectives to be accomplished, possibly with each objective's associated priority/payoff.
- Intentions: The currently chosen course of action.
- Plans: Means of the achieving certain future world state.

The BDI approach has proved valuable for the design of agents that operate in a dynamic environment [28] [29]. Desires are things the agent wants to achieve. They play an important role in the philosophical foundations, but the logical theory deals with goals, which are assumed to be the consistent set of desires. At the implementation level, the motivational concept is deduced to events [30].

A family of BDI systems that can operate in dynamic, volatile, or real-time environments can be constructed by adding an operational semantics associated with a plan that is consistent with notions of bounded or resource limited rationality. These agents implement a form of reasoning where behaviors are pre-planned sequences of goals to be achieved and actions to be taken [31]. A goal could be decomposed into sub-goals and their related sub-plans. A plan provides an abstract specification for the achievement of a goal.

The belief-desire-intention (BDI) approach to agent teamwork has led to many practical multi-agent applications [32] [33] [34].

2.3.8 JADE agent platform

The JADE Agent Platform complies with FIPA specifications and includes all those mandatory components that manage the platform that is the ACC (Agent Communication Channel), the AMS (Agent Management System), and the DF (Directory Facilitator). All agent communication is performed through message passing, where FIPA ACL is the language to represent messages. The standard model of an agent platform defined by FIPA is shown in Figure 2-7.

The agent platform can be distributed on several hosts. Only one Java application, and therefore only one Java Virtual Machine (JVM), is executed on each host. Each JVM is basically a container of agents that provides a complete run time environment for agent execution and allows several agents to concurrently execute on the same host [35]. Figure 2-8 shows an agent platform over three computers. The main container sits on host1. Two agent containers register with the main container through java RMI.

Figure 2-7 Reference architecture of a FIPA Agent Platform

Figure 2-8 JADE Agent Platform over Several Containers

2.4 Capture/replay tools

There are hundreds of automation tools available in the market. Most of the GUI automation tools have a feature called ‘record and playback’ or, ‘capture replay’. Using this feature, the tester executes a test manually. The test tool runs in the background and capture what the tester does and generates a script, which can be run to re-execute the test automatically. Scripts could also be developed in the language used by the tool to run the test cases and execute the test cases in an automated style. Below is a brief introduction to WinRunner and Rational Robot, which are two

leading “capture/replay” tools in the market. Further information could be found in their user manuals.

2.4.1 WinRunner

WinRunner [8] is a functional testing tool that verifies GUI applications work as expected. Users can capture, playback and verify business processes to automatically identify and log defects throughout the application lifecycle. WinRunner has the ability to capture a business process into a script by recording user actions against the application.

It supports a wide array of verification types from text, object, and bitmap to database verification. It can run multiple tests running with varying data maintained in the data table. It also enables testers to take a single script and modify its run-time settings so that it can be replayed correctly against varying screen sizes.

WinRunner automatically identifies functional problems and displays errors after each test.

The WinRunner testing process consists of 6 main phases:

1) Teaching WinRunner the objects in the GUI application

Before running a test, WinRunner first learns how to recognize the objects in GUI applications. It looks at the object’s physical properties and learns the properties that uniquely distinguish an object from all other objects in the application.

WinRunner saves the properties into a GUI map file. There are two modes to create GUI map files. One is per Test mode and the other one is the Global GUI Map File mode. In the GUI Map File per Test mode, WinRunner automatically creates a new GUI map file for every new test created. WinRunner automatically saves and opens the GUI map file that corresponds to a test. In the Global GUI Map File mode, a single GUI map is used for a group of tests.

An object description in the GUI map is composed of: a logical name, a short intuitive name describing the object. This is the name in the test script. For example:

```
button_press ("Query");
```

“Query” is the object’s logical name.

A physical description is a list of properties that uniquely identify the object. For example:

```
{  
  
  class: push_button  
  
  label: "Query"  
  
}
```

When running a test, WinRunner reads an object’s logical name in the test script and refers to its physical description in the GUI map. WinRunner then uses this description to find the object in the application under test. If an object changes in an application, the tester must update its physical description in the GUI map so that WinRunner can find it during the test run.

2) Creating additional test scripts that test GUI application’s functionality

Test scripts could be created either by recording or programming.

By recording, the tester works with the GUI application as usual, clicking objects with the mouse and entering keyboard input while WinRunner runs in the background. WinRunner records operations and generates statements in TSL (Mercury Interactive’s Test Script Language).

Two record modes are available: Context Sensitive and Analog. Context Sensitive mode records operations in terms of the GUI objects in the GUI application. For example, if the tester records a mouse click on the **OK** button then TSL statement `button_press ("OK")` is generated. When running the script, WinRunner reads the command, looks for the **OK** button, and presses it.

In Analog mode, WinRunner records the exact coordinates traveled by the mouse, as well as mouse clicks and keyboard input. When running the test, WinRunner play the recorded movements using absolute screen coordinates. If the GUI application is located in a different position on the desktop, or the user interface has changed, WinRunner is not able to execute the test correctly.

The tester may also add checkpoints at recording time. At playing time the tester can determine whether it is functioning properly according to the behavior of its GUI objects. If a GUI object does not respond to input as expected, a defect probably exists somewhere in the application's code.

3) Debugging the tests

The tester debugs the tests to check that they operate smoothly and without interruption.

4) Running the tests on a new version of the application

The test is run on a new version of the application in order to check the application's behavior.

5) Examining the test results

Examine the test results to point defects in the application.

6) Reporting defects

Once a test has been debugged and run, the same test can perform with multiple sets of data. To do this, the tester needs to convert the test to a data-driven test and create a corresponding data table with the sets of data the tester wants to test.

Converting test to a data-driven test involves the following steps:

- 1) Adding statements to the script that open and close the data table.
- 2) Adding statements and functions to the test so that it will read from the data table and run in a loop while it applies each set of data.

- 3) Replacing fixed values in recorded statements and checkpoint statements with parameters, known as parameterizing the test.

Following is a script example shown in the user manual of WinRunner:

```
set_window("Flight Reservation",3);  
  
menu_select_item("File;Open Order...");  
  
set_window("Open Order",1);  
  
button_set("Order No",ON);  
  
edit_set("Edit_1","3");  
  
button_press("OK");
```

2.4.2 Rational Robot

IBM Rational Software's Rational Robot [9] is a similar GUI automation tool to WinRunner. It allows one to record and replay test scripts that recognize the objects in various applications, and view and edit them while recording.

The Rational Robot testing process consists of five main phases:

- 1) Create a Rational repository with the Rational Administrator to store the test assets.
- 2) Record and play back scripts in Rational Robot.
- 3) Review the results of playback in the Rational LogViewer and Comparators.
- 4) Create defects about any failures in the scripts with Rational ClearQuest.
- 5) Analyze the results of the test by creating reports in TestManager, LogViewer and ClearQuest.

When recording a script, Rational Robot uses its Object-Oriented-Recording technologies, which identifies objects by their internal object names, to record both the actions as the tester navigates through the application under test.

During recording, the tester can insert one or more verification points in a script to capture and store information about the objects that is being tested.

Robot could find and tests them during playback. If there are changes, Robot will mark them and let the tester decide whether they are enhancements or defects in the application.

Following is a script example created by Rational Robot.

```
Sub Main
```

```
StartApplication "notepad"
```

```
Window SetContext, "Caption={* - Notepad}", ""
```

```
MenuSelect "File->Open..."
```

```
Window SetContext, "Caption=Open", ""
```

```
InputKeys "C:\Names.txt"
```

```
Pushbutton Click, "ObjectIndex=1"
```

```
Window SetContext, "Caption={* - Notepad}", ""
```

```
Result=EditBoxVP(CompareText,"ObjectIndex=1","VP=TextVerification;Type=CaseSensitive"  
)
```

```
Window SetTestContext, "Caption=Names.txt - Notepad", ""
```

```
Result=EditBoxVP(CompareText,"ObjectIndex=1","VP=TextVerification;Type=CaseSensitive"  
)
```

```
Window ResetTestContext, "", ""
```

```
Window SetContext, "Caption=Names.txt - Notepad", ""
```

```
MenuSelect "File->Exit"
```

```
End Sub
```


2.5 Summary

Characteristics of GUI application, concepts of agent and Expert System technology are discussed in this chapter. This chapter also gives brief introductions to two leading “capture/replay” tools. GUI applications are event driven applications and their GUI consists of GUI elements, which could be identified by capture tools. The features of a software agent are autonomous. An intelligent software agent is a software agent that uses AI. Agents may interact with another agent to achieve a goal. Expert system is one of a group of disciplines in AI. Expert system shell has the ability to reason about a set of loaded rules and the facts. “Capture/replay” tools record user’s actions into script files and replays it at a later time. These tools can only replay exactly what has recorded.

Chapter 3 Intelligent Agent Based Automatic Operating Model

This chapter discusses a general model for intelligent agent based automation. Based on this model, two elaborated models are presented in Chapter 4 and 5.

Today, most operating systems such as Microsoft Windows, Mac OS, Linux, Sun Solaris and HP UNIX, provide a graphical user interface. Human activities have also been provided an alternative approach via GUI. We work, study, communicate and even shop through GUIs. Thus there is a great need to make these activities more convenient by automating the operations over the GUIs. The needs for automation are arisen now and then. It is extremely inflexible and expensive to request a new feature of the information system whenever a need arises, if at all possible. In fact, given the time constraints, and cost and availability of solution providers, it is often not practical.

At present, there are no effective techniques for GUI application automation. Currently, there are two ways to automate a GUI application. One way is that a program is written to simulate the mouse click and keyboard input. This way is flexible, but the user must possess programming skills. In real life, it is not realistic to expect that the user of a GUI application have such skills. Therefore, this way has only limited adoption. The other way is to use the “capture/replay” tools to capture all user interactions with the GUI application and stores these actions into a script file. These actions include keystrokes, mouse movements, menu selections and button clicks. At a later time, when the tool replays the recorded script, it initiates the same actions that the user performed.

Using the “capture/replay” approach to automate GUI application, it is limited to replaying exactly what has been recorded. When automating a product application, it lacks the ability to play different actions according to the execution context of the running GUI application. Hence, it lacks intelligence. For example the “capture/replay” tool could not delete emails received before a specific date. When automating an application under test, it lacks the ability to adapt to changes in the GUI application. A

modification to the GUI may cause some of its scripts to become invalid and need to be discarded or re-captured.

This chapter discusses an automatic operation model that can be elaborated and applied to two types of activity automations through GUIs. The first type is performed over interfaces of GUI applications released as products. These applications have passed the test phase and been released as product applications (PA) by the software vendors. End-users operate these applications to perform various tasks. Unless the users buy new releases of the applications, they are considered as constant applications, i.e. no more changes will be made to the applications.

The second type of activities is performed over interfaces of applications under test (AUT). They are operated by testers to identify possible bugs. These applications are still in the development phase and are subjected to frequent change. The changes may be caused by new requirements or bug fixes.

GUIs are used by users to interact with its applications. Since logics are not explicitly presented on the GUIs, the automation relies on actionable knowledge of the users or testers when they operate the GUI applications.

The proposed automatic operating model takes a new perspective of GUI knowledge perspective. It uses knowledge to represent the underlying implied logic of GUI applications and applies software agents to carry the knowledge to automate GUIs.

Section 3.1 discusses the definition of GUI model. Section 3.2 discusses the intelligent agent based automatic operating model. Section 3.3 discusses the automation of two types of activities through GUIs.

3.1 Definition of GUI model

A GUI enables the user to operate the GUI using the mouse or to enter arbitrary alphanumerical values into edit boxes. GUI makes the applications much easier to learn and use [3]. It is also simpler to visualize the progress of data processing and to give the necessary background information. Due to these reasons, GUIs have become critical components of today's software [36].

Chapter 3 Intelligent Agent Based Automatic Operating Model

A GUI application is composed of windows. Each GUI object has many properties, such as “class”, “label”, “width”, “height”, “handle”, “enabled”, etc. and each element can be uniquely distinguished by part or all of its properties.

In [37], a GUI is modeled as a set of widgets $W = \{w_1, w_2, \dots, w_l\}$ (e.g., buttons, text box) that constitute the GUI, a set of properties $P = \{p_1, p_2, \dots, p_m\}$ (e.g., background color, size, font) of these widgets, and a set of values $V = \{v_1, v_2, \dots, v_n\}$ (e.g., red, bold, 16pt) associated with the properties. Each GUI will contain certain types of widgets with associated properties. At any point during its execution, the GUI can be described in terms of the specific widgets that it currently contains and the values of their properties. Some of the widgets with associated values could be used as the identifier of GUI **states**.

At the same time, multiple GUI applications share the same computer’s graphical display presentation resources of a windowing system. A windowing system is an event driven system. Each hardware and user action causes one or more messages to be generated. Windows events are communicated via messages. A number of windows from one or more GUI applications may be displayed on the screen at the same time, but only one window has the focus.

The GUI responds to user events, such as mouse movements or menu selections, providing a front end to the underlying application code. The GUI interacts with the underlying code through messages or method calls [38].

The classification of GUI events that we employ in this thesis is as follows:

- System interaction events interact with the underlying software to perform actions, such as Change Event which indicates the property value of a GUI element has changed.
- User interaction events are events triggered by the user. Normally these events are mouse and keyboard related events. Such as Click Event, DblClick Event, MouseMove events, etc.

Since we need to capture user’s actions, only user interaction events are used.

3.2 Intelligent agent based automatic operating model

Recent solutions [39] have shown that multi-agent system (MAS)-based approaches are well suited to resolve complex problems. With this multiple intelligent software agents approach, a large problem can be decomposed into several smaller sub-problems and then be solved by a group of software agents [40].

To automate a GUI application, the user's knowledge of operating a GUI application needs to be captured first. This knowledge includes the user's actions, which include the mouse and keyboard actions, and the GUI context that causes this action. The GUI context is the property values of one or more GUI elements.

The general model integrates the human user's knowledge on how to operate the elements provided by the computer interfaces, especially GUIs for a particular goal. It is modeled as a 3-tuple $\langle A, K, E \rangle$ where:

1. A is a set of agents $A = \{ag_1, ag_2, \dots\}$, where ag_i is an agent.
2. E is the execution context.
3. K is a 3-tuple $K = \langle AC, ST, CS \rangle$, where AC is a set of GUI actions, ST is a set of states and CS is a set of constraints applied to AC and ST . The relationship between the states and actions could be defined as: $y = f_i(x_1, x_2, x_3, \dots, x_i)$ where $x_i \in ST, i \in N, y \in AC, f_i \in CS$

In this model, agents may include GUI states monitoring agents, input data reading agent, GUI action playing agent, etc.

The execution context is represented as a set of states $S = \{s_1, s_2, s_3, \dots\}$ (window caption, error message, number of input data, etc.) that affect the agent's action on the GUI and the agent's action may affect these states. A state value is returned from a state function which is provided by the implementation. Such as function "getWindowTitle" is used to get the title of a window.

The automatic operating model is depicted in Figure 3-1.

Figure 3-1 General intelligent agent based automatic operating model

The process of the automation is divided into two steps: the knowledge capture step and automation step.

To automate a GUI application, the first step is to capture the knowledge of the user when he operates the GUI application and represent the captured knowledge in the knowledge base.

The second step is to automate a task with the help of the agents on behalf of the user. These agents operate the GUI application automatically using the captured knowledge.

3.3 Two types of activities automation

The purpose of operating a product application is to achieve a specific task. It should have the ability to take different actions based on the execution context of the running GUI application. The purpose of operating an application under test is to find bugs. It checks if the running GUI application conforms to the design specification. Because of the different purposes of operation of product applications and applications under test, two elaborated models are derived from the general model: intelligent agent based operating model for product applications and intelligent agent based operating model for regression testing.

3.3.1 Intelligent agent based operating model for product applications

With GUI applications, we can perform numerous tasks. For example, we can use a web browser to read news. We can also use a web browser to send and receive emails.

Chapter 3 Intelligent Agent Based Automatic Operating Model

The purpose of GUI automation is to use a GUI application to achieve a task on behalf of the user. Hence, the user's knowledge on how to operate the GUI application needs to be captured. The captured knowledge is then used to achieve different types of tasks automatically. At a later time, the user just specifies the name of the task to be achieved. The task is then performed automatically by the agents.

The task can also be automated by writing a program to operate a GUI application. But writing a program to automate a GUI application needs complex programming skills, and may not be adopted widely.

To automate GUI applications, based on the general model, in Chapter 4, we propose an agent based model for the automation of product GUI applications.

In this model, the knowledge $K=\langle AC, ST, CS \rangle$ is defined as below:

AC is a set of GUI actions, ST is a set of states including GUI states and input data states, and CS is a set of rules. The "if" part of the rule is the states in ST , and the "then" part of the rule is a GUI action in AC .

In this model, five agents are identified to do the GUI automation: user interface agent, playing agent, monitoring agent, reasoning agent and input data reading agent. At capturing phase, a capturing application is used to record user's actions and convert these actions into rules. At automation phase the user assigns tasks through the user interface agent. The playing agent is used to re-generate mouse and keyboard events and send them to the underlying operating system. The monitoring agent is used to monitor the GUI states of an application. The input data reading agent is used to provide input data. The reasoning agent is used to do the reasoning work in response to the change of states in the execution environment perceived by the monitoring or input data reading agent. These agents work together to drive a GUI application to achieve a specific task automatically.

If the user is operating a GUI application not for GUI testing then the actions belong to this activity.

In Chapter 4, the detailed description of the model is presented.

3.3.2 Intelligent agent based operating model for regression testing

The widespread use of GUIs has led to the construction of more and more complex GUIs. GUIs are pervasive in today's software systems and constitute as much as half of software codes [41] [42]. Although they make software easy to use from a user's perspective, they largely complicate the software development process [43] [44] [45]. The use of GUIs in safety-critical systems is also growing, making their correct operation imperative [38].

The development of GUI-based applications has raised a lot of new issues, one of them is how to effectively test complicated graphical user interactions [46].

GUI testing requires that test cases (sequences of GUI *events* that exercise GUI widgets) be generated and executed on the GUI [47]. Current GUI testing techniques are incomplete [47]. Recent advances in GUI testing have focused on the development of test case generators [48] [49] [20] [50] [4] [51] and test oracles [52] for GUIs. Even though a few automated GUI test case generation schemes have been proposed, in practice, test cases are still being generated manually using capture/replay tools [53].

Regression testing is a testing process that is used to determine if a modified program still meets its specifications or if new errors have been introduced [7]. Regression testing is especially important for GUIs since GUI development typically uses a rapid prototyping model [54] [55] [56] [47]. Even when tools are used to generate GUIs automatically [41] [57] [58], these tools themselves may contain errors that may manifest themselves in the generated GUI, leading to software failures. Hence, this development environment requires efficient regression testing mechanisms that can detect the frequent software modifications [47].

The characteristics of GUIs present special challenges when verifying a GUI's behavior [59] [43] [44]. The GUI testing problem is difficult and challenging [4] because:

- The input space is extremely large due to different permutations of inputs and events, and complex GUI dependencies may exist [60].

Chapter 3 Intelligent Agent Based Automatic Operating Model

- The large number of possible states results in a large number of input permutations [61].
- There is no specific output. Each action affects the state of the GUI [5].

Although regression testing [62] [63] [64] [65] is an important software maintenance activity for traditional software, accounting for as much as one-third of the total cost of software production [66] [67], regression testing of GUIs has remained a largely unexplored area [1].

Automated testing involves executing test cases and verifying the result programmatically instead of relying on human ability. Test automation is a software development that can be thought of as “writing software to test other software” [2]. Automated testing has the ability to reduce testing costs, because automated tests can execute test cases much faster than manual testing [68].

Although “capture/replay” tools are used to automate GUI regression testing [47], the approach suffers the lack of flexibility because we can only replay whatever we have recorded. Slight changes to the application or the test case may invalidate the recorded test case [46]. Test cases that cannot be rerun simply discarded [1]. The problem of unusable test cases is especially serious for GUIs, since GUI modifications make a large number of test cases unusable, requiring expensive regeneration.

To automate GUI regression testing, based on the general model, in Chapter 5, we propose an agent based model for the GUI application regression testing.

In this model, the knowledge $K = \langle AC, ST, CS \rangle$ is defined as below:

AC is a set of GUI actions. ST is a set of GUI states. CS is a set of order constraints between two elements in AC and ST . In this model, the actions and states are represented as an actionable knowledge graph (AKG).

In this model, seven agents are identified: the user interface agent, the capturing agent, the playing agent, the remote screen capturing agent, the remote controlling agent, the KB managing agent and the mediating agent. The user interface agent is used to accept

Chapter 3 Intelligent Agent Based Automatic Operating Model

instructions from the tester and displays information generated by the mediating agent. The capturing agent is used to capture the tester's actions when the tester operates the GUI application under test. The playing agent is used to perform the GUI actions of a test case. The remote screen capturing agent is used to capture the screen image of a remote GUI application and send them to the remote controlling agent. The remote controlling agent is used by the tester to display the remote screens and to control the remote GUIs. The KB managing agent is used to convert, search and reconstruct test cases. The mediating agent plays the role of mediator in the system. It processes the tester's commands, relays messages between other agents, and keeps log information.

If the user is doing regression testing, the actions belong to this activity.

3.4 Summary

In this chapter we discussed the general intelligent agent based automatic operating model. We also discussed the two types of GUI applications to be automated. Two elaborative models based on the general model were also discussed briefly. The two derived models have different knowledge representation and composite agents. Agents in the two models are identified based on the roles they play.

Chapter 4 Automatic Operating Model for Product Applications

In this chapter, we propose an intelligent agent based model to operate GUI applications automatically. Section 4.1 gives the overview of our research on GUI automation. Section 4.2 is the architecture of the proposed automatic operating model. Section 4.3 describes the capturing application. Section 4.4 is the proposed rule presentation of GUI actions and execution context. Section 4.5 defines the agents used in this model. Section 4.6 describes interactions between agents. Section 4.7 is the summary.

4.1 Overview

GUI application **automation** is to achieve a specific task by operating a GUI application automatically with the help of another application(s) on behalf of the user. It allows a GUI application to be operated automatically without the user's intervention.

From Chapter 2, we know GUI applications are interactive applications. They act in response to the user's actions, which include mouse clicks and keyboard input. To achieve a task using a GUI application, user's involvement is needed.

First, we investigate the characteristics of operating a GUI application manually.

To be able to operate a GUI application to perform a specific task successfully, the following requirements of a user are needed:

- 1) The user must have a good idea on what he/she is going to do with the GUI application. We call it a **task**.
- 2) The user knows how to use the GUI application to achieve his/her task. In other words, the user must own sufficient knowledge about the GUI application.
- 3) Have prepared input data if the GUI application requires any input data.

To achieve a task, the user first needs to start the GUI application. Then based on his knowledge about the GUI application, the user begins to operate the GUI application by using mouse and keyboard. After each action, the user needs to check the screen to get the state of the GUI application. The user takes further actions based on the response of the GUI application. Exceptions may occur during the execution of the GUI application. For example, the printer becomes unavailable when the user tries to print a document from Microsoft Word. If this happens, the user may need to retry or use another configured printer in the system.

Factors that affect the user's decision of what action is to be performed next are listed as follow:

- 1) States of input data. If there are more input data are needed to be inputted, the actions related to an input data form need to be performed repeatedly. For example, if we have more staffs to be registered with a system, then we need to repeat the actions for registering one staff.
- 2) States of the GUI application (context). There are two kinds of states with a GUI application: **expected GUI states** and **unexpected GUI states**. **Expected GUI state** is the normal execution state (output) of a GUI application, which is reflected on the screen. The user takes different actions based on the values of the expected states. For example, when the user checks his/her bank transactions, if his/her task is to delete transactions made last year, then the "date" field on the GUI of each transaction becomes an expected state. **Unexpected GUI states** are the states that the user doesn't expect them to happen when he/she is achieving his/her task. But the user needs to deal with these unexpected states before carrying on the current task. Unexpected states may occur at any time during the execution of a GUI application. Unexpected states are caused by the current action or by the system. For example, the internal server error or connection refused error occurs when user is browsing the Internet.

We define **states** as the information that affect the user's further actions on GUIs of a GUI application. The information includes significant messages which are shown on the GUI or input data that needs to be input. Significance means that these states are

Chapter 4 Automatic Operating Model for Product Applications

sufficient enough for the user to make a decision about which next GUI action is to be taken.

To automate a GUI application on behalf of a user, we need to capture the user's knowledge when he/she is performing a task. We need to capture both the user's actions as well as the states that cause the actions.

Most of the commercial "capture/replay" tools have shown the following technologies:

- 1) Read messages displayed on the screen.
- 2) Simulate keyboard and mouse events.
- 3) Capture user's actions in component level and stored these actions using scripts.

This thesis assumes that the above technologies are ready and available to be used.

Since the "capture/replay" approach lacks reasoning ability, in this chapter, we propose an intelligent agent based operation model to automate GUI application. In this model, five agents are identified to do the GUI automation: user interface agent, playing agent, monitoring agent, reasoning agent and input data reading agent. At capturing phase, a capturing application is used to record user's actions and convert theses actions into rules. At automation phase, the user assigns tasks through the user interface agent. The playing agent is used to re-generate mouse and keyboard events and send them to the underlying operating system. The monitoring agent is used to monitor the GUI states of an application. The input data reading agent is used to provide input data. The reasoning agent is used to do the reasoning work in response to the change of states of the execution environment perceived by monitoring and the input data reading agent. These agents work together to drive a GUI application to achieve a specific task automatically.

The contribution of this chapter is shown as following:

- 1) An intelligent agent-based model to automate GUI applications. In this model, the user's actions and the execution context of a GUI action are stored in the form of knowledge, which is represented as rules and facts, instead of scripts.

Thus, different GUI states and input data states will trigger different rules and play different actions.

- 2) The model is able to process unexpected GUI states.
- 3) The captured input data and GUI states are extracted automatically and stored in text file format. It is easy for the user to modify the recorded input data and GUI states and reuse the rules without any programming.

As we have discussed, many human activities have largely converged to computer interfaces. Therefore, the automatic model has wide potential in numerous areas, such as personal digital assistants, software testing, e-learning, e-business and etc...

GUI applications are event driven applications. Whenever an event happens, which is either caused by the windowing system or by the user, such as a mouse or keyboard event, the GUI application will react according to the happened event. Based on the changed GUI states, the user could take further actions. This process is very similar to the behavior of a rule forward chaining approach in an expert system. It is suitable to represent the GUI applications context as the “if” part of a rule, and the user’s actions as the “then” part of a rule.

The inference engine separates the rules from the other part of the system, and rules in an expert system are treated as unordered. New rules for a task can be added to the existing rule file easily by just appending the new rules to the rule file of a task. Hence, the knowledge about a GUI could be accumulated and there is no technical requirement from the user. However in the “capture/replay” method, we need to rerecord everything from the beginning or manually modify the recorded script file to insert these actions to a specific position.

Since the rule engine does the pattern matching, we do not need to write our own states machine processing program. Further, because rules in an expert system are treated as unordered, therefore if the “if” part of rule is satisfied, this rule will be fired by the rule engine. Sequential discipline needed in conventional programming does not exist any more. This is especially useful in processing unexpected states because the unexpected exception could happen at any time and the types of unexpected exception could not be

known in advance. If we use the state machine method, then whenever a new unexpected exception has happened, we need to modify the state machine and link all the existing states to this new state. As more unexpected exceptions happen, the state machine will become complex and hard to maintain. However when a new unexpected exception is met by using the expert system technology, we just need to generate the rules for the action which is caused by the unexpected states. At automation phase, this rule will be triggered automatically without any extra programming effort when this unexpected state appears. It is productive and the effort of processing an unexpected state is simplified. So whenever a new unexpected state is met, the user's actions in processing the unexpected state are captured and converted into rules automatically. In this model, processing of an unexpected state is treated as a task.

States are gotten through **state functions**. The **state functions** are used to get GUI state values or input data values at automation time.

To automate a GUI application, first we need to capture the user's knowledge and represent them in the knowledge format. The knowledge is stored in four types of file: unexpected states file, expected states file, rules and facts file, and input data file.

At the capturing phase, the user needs to assign a task name to start the automation. When a task is finished, the above four files are generated for this task. After that, the user could capture another task.

Here, it may argue that the "capture/reply" approach also stores user actions in a script file. So what's the difference? The difference is that in the "capture/replay" approach, unless the user manually modifies the script file, such as adding "if-then" and "loop" statement provided by the script language, the script file is a sequence of actions in the order when they are captured. It is illustrated in Figure 4-1.

In Figure 4-1, action a_1 happens before action a_2 and action a_2 happens before action a_3 . Using the "capture/replay" approach, at play time, the sequence of actions to be performed is $a_1, a_2, \dots$.

Chapter 4 Automatic Operating Model for Product Applications

This means that the play order of an action a_i relies on position they are stored in the script file. Therefore, a_1 is always executed before a_2 . To alter the execution sequence, programming is needed.

File Head

$a_1 \ a_2 \ a_3 \dots \dots \dots a_n \dots \dots \dots$

 File Tail

Figure 4-1 Actions in a script file

However in this model, we don't only capture the relationship between the actions, but also the relationship between the states (GUI states and input data states) and the actions. These relationships are represented as rules. The trigger of a rule is based on the GUI states, not the order that they are stored in the file.

At capturing phase, to manually assign each action a set of states is not practical. For example, when we use Microsoft Outlook to read the first email in the inbox folder, if we find that it is from "sg@jobstreet.com" then we open it. Otherwise we delete this email. The text value in the first field of column "From" is identified by the user as the state value for actions of opening the mail or deleting the mail. For other actions in the "reading email" task, the completion of a previous GUI action is used as the default state. Figure 4-2 is the screen shot of Microsoft Outlook.

Figure 4-2 Screen shot of Microsoft Outlook

Figure 4-3 shows the actions flow in task "reading email".

Chapter 4 Automatic Operating Model for Product Applications

Figure 4-3 Sample action flow of reading email

Action A represents the open of the email application (here to be simple, we use a single *Action A* represents the one which opens the Microsoft Outlook). *Action B* represents the clicking on the “Inbox” button. “*get_text*” is a state function which reads text from the screen. State value S_1 represents “=sg@jobstreet.com” (equal to sg@jobstreet.com) and S_2 represents “!=sg@jobstreet.com” (not equal to sg@jobstreet.com). *Action C* represents the clicking on the first mail item. *Action D* represents the deletion of an email. *Action E* represents the clicking on the “x” button of the outlook to exit. The recording application records the user’s actions and allows the user to insert state values for the immediate following action of these state values. The recording application provides a set of state functions for the user to use. These state functions help the user to capture state values from the screen. How a state function works depends on the implementation. It should returned the state value and let the user to modify the value. The user first starts the capturing application and gives the task name “read email” to the capturing application. Then he performs *action A*, *action B*. These actions are captured by the capturing application. The execution time of each action is also captured by the capturing application. A state action “delay *action_name* *execution_time*” is generated automatically by the capturing application for *action A* and *action B*. After *action B* is complete, before performing *action C*, if the user wants to insert a state value that indicates *action C* is performed when a mail is from “sg@jobstreet.com”, then the user selects function “*get_text*” provided by the capturing application and clicks on the first field of the “From” column. State function “*get_text*” returns value “sg@jobstreet.com”. Then the user modifies the state value to “=sg@jobstreet.com”. The capturing agent then generates a state action “*get_text*”

Chapter 4 Automatic Operating Model for Product Applications

From(0) =sg@jobstreet.com”. Here “From(0)” means the first row of column “From”. The user performs *Action C*. After *Action C* is complete, *Action E* is performed. The following rules are produced by the capturing application:

- 1) if “Starting” is true then executes action A. // here we use “Starting” as the state value of the first action in a task.
- 2) if Action A is complete then executes action B.
- 3) if Action B is complete and it is from “sg@jobstreet.com” then executes action C.
- 4) if Action C is complete then executes action E.

At a later time if the user wants to capture the knowledge of deleting the mails which are not from “sg@jobstreet.com”, he could use the capturing application to add this knowledge to the existing task “read email”. To add this knowledge, the user first performs *action B* and tells the capturing application the next action to be performed using the following steps: After *action B* is complete, the user selects state function “get_text” provided by the capturing application and clicks on first field of “From” column. Function “get_text” returns value “Yin Zun Liang”. The user then modifies this value to “!=sg@jobstreet.com” and performs *Action D*. After *Action D* is complete, *Action E* is performed. The following rules are produced by the capturing application:

- 5) if Action B is complete and the sender is “!=sg@jobstreet.com” then executes action D.
- 6) if Action D is complete then executes action E.

Because rules are generated after each action, so the user could add new knowledge to an existing task easily.

From the above example, we can see that the execution order of actions is embedded in the rules. It does not rely on the position the rules appearing in the rule file. If *rule 1* and *rule 6* change their position in the file, it does not affect the result of the task. Furthermore, at automation phase *rule 3* and *rule 5* could be triggered accordingly according to

the name of the sender without any programming. In the “capture/replay” approach, if *rule 3* and *rule 5* are actions then they are both executed and *rule 3* must be performed first.

We use the following example to illustrate how to use rules to represent repeated work.

Here, we also use the Outlook example. If the user wants to delete all the emails, he first opens the Microsoft outlook, and then clicks the “Inbox” button to open the inbox folder. Then he selects “*get_text*” to check if there are emails to be deleted. If “*get_text*” returns a none-null value string, he presses “delete” button to delete this email. If “*get_text*” returns a null value string, the user clicks on the “x” button of the Outlook to exit. After deleting an email, the user selects “*get_text*” to see if there are more emails to be deleted. If “*get_text*” returns a null value string, the user clicks on the “x” button of the outlook to exit. If “*get_text*” returns a none-null value string, he presses “delete” button to delete this email. Figure 4-4 shows the actions flow. *Action A* represents the opening of Microsoft Outlook action. *Action B* represents the opening of “inbox” folder. State value S_1 represents “!=null” and State value S_2 represents “=null”. *Action C* represents the “delete” action. State S_3 represents “=null” and state S_4 represents “!=null”. *Action D* represents clicking on the “x” button of the Outlook to exit.

Figure 4-4 example of repeated work

The following rules are produced by the capturing application:

Chapter 4 Automatic Operating Model for Product Applications

- 1) *if “Starting” is true then executes action A*
- 2) *if Action A is complete then executes action B*
- 3) *if Action B is complete and there are emails then executes action C*
- 4) *if Action B is complete and there are no emails then executes action D*
- 5) *if Action C is complete and there are emails then executes action C*
- 6) *if Action C is complete and there are no emails then executes action D*

From above we could see, the six rules could represent deleting all emails and stopping when there are no emails in the “Inbox” folder. If we use the “capture/replay” approach, if the user deletes three emails at recording time, then at playing time, it could only delete three emails exactly. If there is no email, it will fail.

To capture the user’s knowledge correctly, the user needs to “train” the capture application. The user is able to add new knowledge to a task whenever a new use case is met. The user is also able to train the capture application with all the cases at one time if the user could simulate the all cases. Since the user is able to modify the state value returned by a state function, the user could manually define the state values. This enables new knowledge is added without needing the real case happens.

At automation phase, unexpected exceptions could happen. When an unexpected exception happens, the monitoring agent could not get the GUI state correctly then no rule will be fired from the reasoning agent. After the predefined time out, the reasoning agent assumes an error has happened and the current task is failed. The failed message will be displayed on the GUI of the interface agent. When this happens, the user needs to use the capturing application to capture all the actions which the user performs during the process of the unexpected exception. Then next time the system could process this unexpected exception automatically. The processing of the unexpected exceptions is also treated as a task. Other unexpected exceptions may occur during the processing of the unexpected exceptions. To process these exceptions correctly, we use the same technology as the interrupt processing technology found in an operating system. When the system detects an unexpected state that happens, it first saves the

Chapter 4 Automatic Operating Model for Product Applications

execution context into stack and begins to process the unexpected exception as a task. After the exception is processed, it resumes the saved context and continues to carry on the interrupted automation. The process is illustrated in Figure 4-5.

Figure 4-5 Exception process

The reasoning agent keeps a timeout value to detect errors. When it does not receive any GUI states from the monitoring agent for the timeout value, it will assume an error has happened and notify the interface agent.

4.2 System architecture

Figure 4-6 Agent Based GUI automation model architecture

The architecture of the proposed intelligent based GUI automation model is shown in Figure 4-6.

In this model, the interface agent is used by the user to enter a new task to be achieved.

At recording time, the capturing application is used to capture the user's actions and convert these actions into rules and facts, and store them into the rules & facts file. GUI states (expected and unexpected states) and input data within the rules are extracted from the rules and stored into different text files. Expected states and unexpected states are stored in expected and unexpected states files respectively. Input data are stored in the input data file.

The reasoning agent is used to control the execution of a task based on the information obtained from the monitoring agent and the input data reading agent.

The playing agent is used to execute the GUI related actions sent by the reasoning agent.

The monitoring agent is used to monitor the states of the GUI after a GUI action, which is sent by the reasoning agent, has been performed. It also monitors all the unexpected states, which are associated with a task.

The input data reading agent is used to provide input data to the reasoning agent as input data states. This requires checking if there is more than one input data in an input data form. The detailed information about input data form is described in Section 4.3.

4.3 Capturing application

The capturing application is used to capture the user's actions and GUI states. If there is an input data in a GUI action, the capturing application also extract the input data and convert them into rules. The format of expected states, unexpected states file are same. The difference is that they hold different kinds of state values.

The internal structure of the capturing application is depicted in Figure 4-7.

Chapter 4 Automatic Operating Model for Product Applications

The capturing application has a GUI that enables the user to send commands to control the capturing application and display response information. The function of each component in the capturing application is as following:

The *controller* is used to control the start, stop and pause of the *event capturer* and the *state capturer*.

The *event capturer* is used to capture the user's mouse and keyboard events in the component level. For example, if the user clicks on the "new" menu item, it will generate an "app1.windowX.New.click" event. This is used as the action name in the monitoring rule and action rule.

The *states capturer* provides a set of state functions to the user and the user could select and insert GUI state functions or input data functions after a GUI action is executed. The state values are stored as the state function's parameter. A state function has two parameters. The first parameter of a GUI state function is always an action name which is a GUI element name with the application name and its parent windows name as prefix. When the user clicks on a GUI element, the context name will be generated automatically. The capturing application will find the parent's window name of the GUI element and prefix it to the GUI element name. The second parameter is the state value. For example, if the user wants to get the title of a window, "getTtitle app1.winA.title A" is generated. The default state function of all GUI actions is the "delay" function, which the second parameter represents the execution time of a GUI action.

Chapter 4 Automatic Operating Model for Product Applications

Figure 4-7 Components in the capturing application

The *actions processor* is used to convert the GUI action and related states into rules & facts. The conversion algorithm is simple. It is shown as below:

- 1) Before a GUI action is to be performed, if the user has selected to capture states then remember the states functions and states value defined by the user.
- 2) If the GUI action is being performed, count the execution time.
- 3) If the GUI action is complete. First generate a state action "*delay action_name execution_time*". States function "delay" is always used for all GUI actions as the state function. Secondly, fill the fields in the monitoring rule, action rule and facts template using the previously saved state actions and "delay" state action. Rules and facts are saved into rules and facts file.
- 4) Save the state values to the expected state file. Save the input data to the input data file. If the task is processing an unexpected state, then save the state value of first action to the unexpected state file.

Chapter 4 Automatic Operating Model for Product Applications

The input data of a GUI application is organized as **input data form**. Each input data form consists of a set of input GUI widgets for the user to input data, such as text box, radio button, etc. We call these inputs GUI widgets as **input fields**. It is the user's responsibility to give a form name and decide which GUI input widgets belong to this form at capturing phase. The user needs to tell the capturing application the action that begins an input form, and the action that ends the input form. An example of GUI application to be automated is shown in Figure 4-8. This GUI application is used to add, delete and query staff information. When the user wants to add a new staff, he clicks on the "New" button to input the staff's name and age. He then clicks "Save" button to save the input data. When the user wants to query an existing staff's information, he clicks "Next" button to display the staff's information. He is also able to click on "Delete" button to delete a staff from the system. The "Exit" button is used to exit the application. The input data form comprises of "name", "age", "email" and "department" input fields and a form name "New-Staff-Frm" that is given by the user. When the user adds a new staff "John" to the system, after having clicked on the "New" button, the user needs to tell the capturing application that the next action ("Name") is the first input field of the form "New-Staff-Frm". From then on, the names of all the input fields are prefixed with the form name. After having entered the age, the user needs to tell the capturing application that the next action ("Department") is the last input field of the form.

Figure 4-8 Example GUI application used for GUI automation

Chapter 4 Automatic Operating Model for Product Applications

The input data is stored in the input data file, which is generated by the capturing application. The following is an example of such an input data file:

Form Start New-Staff-Frm

New-Staff-Frm.name=John

New-Staff-Frm.age=31

New-Staff-Frm.email="aaa@en.com"

New-Staff-Frm.department="HR"

Form End New-Staff-Frm

Form Start New-Staff-Frm

New-Staff-Frm.name=luc

New-Staff-Frm.age=28

New-Staff-Frm.email="bbb@en.com"

New-Staff-Frm.department="EEE"

Form End New-Staff-Frm

Besides the GUI states, the user is also able to check the input data in a specific input data form after a GUI action has been performed. We define a state function “check_data” for the user to check the input data state, which takes a form name as its parameter. State function “check_data” is used to check whether all the data of a form has been read. The input data reading agent keeps a pointer for each input data form. If the pointer reaches the last “Form End” element of a set of continuous input data form elements, which have the same name, a “false” value will be returned by the “check_data” function. After a GUI action has been performed, the user may insert the function “check_data” as a state function. These GUI actions should not be any input field actions inside an input data form. In that case, the capturing application gets the next input data, which belongs to an input form, by using “next_data” function automatically. Function “next_data” takes a form name as its parameter. A null value will be returned by “next_data” function if the pointer of this form reaches any “Form

End” element of a form. This makes it possible to support optional input data and repeatable input data set.

Expected GUI states are used to control the execution flow of a task. For example, when a user is checking staffs’ information by clicking on the “Next” button, he will delete the staff by clicking the “Delete” button if the age of the staff is greater than or equal to 80. Otherwise, he clicks the “Next” button to view other staffs’ information. The input data may also affect the user’s action flow, for example, when the user adds new staffs information. After having clicked the “Save” button”, the user add a data state function to get the data state by inserting a “check_data” function. Then he clicks on “New” button and tells the capturing application that this action is performed if the “check_data” function returns true. Otherwise, he clicks on the “Exit” button and tell the capturing application that this action is performed when the “check_data” function returns false. The user needs to add state functions explicitly if the user wants to take different actions based on the GUI states and input data state. If no state function is added to a GUI action, a “delay” state function is added automatically with the executing time of the GUI action as its parameter.

State functions are provided by the *states capturer*. In this model three state functions are predefined: “next_data”, “check_data” and “delay”. The more state functions provided by the capturing application, the more states the system could check at automation phase. To really automate a task, a “get_text” state function is suggested to be provided. State function “get_text” is able to scrape the text from a GUI element, such a button, a menu item, etc. if the function “get_text” is provided, then the agents could take different actions based on the text value of a GUI element.

For each GUI state value that the user specified for a state function, an operator may be added before the state value. The operators could be: “=, >, >=, <=, <, !=”. There could be more operators as long as they are consistently interpreted in the capturing application and the reasoning agent. The operators are used by the reasoning agent to assert a state value to its rule engine. For example, if the state value of age captured by the monitoring agent is “81” and the user defined it as “>80” at capturing phase, the reasoning agent will assert the fact “State-value” with slot value “>80” instead of “81”. Otherwise the pre-captured rule won’t work.

Rules are generated for each action. This includes a monitoring rule that retrieves all the expected states functions defined by the user, and an action rule that retrieves a GUI action. Detailed information is further described in next section.

4.4 Rules representation

In this model, two rules are generated for each action by the capturing application after an action is performed and the user has inserted state functions. The two rules are the monitoring rule and the action rule. A monitoring rule is also generated for the task. It defines the states for the task that need to be triggered.

The action rule is defined as following:

IF: the GUI states and the input data state match the GUI action's activation states which is specified at capturing phase

THEN: execute function *play_action*

The expression of the action rule is shown as following:

$$GA(X) = \text{GUI-action}(X)$$

$$A(AG) = \text{Agent}(AG)$$

$$SF(A, F) = \text{State-function}(A, F)$$

$$SV(F, V) = \text{State-value}(F, V)$$

$$AV(X, AG, AN, F, V) = \text{Activation-value}(X, AG, AN, F, V)$$

$$PA(AC) = \text{Play-action}(AC)$$

$$\forall X, A_1, F_1, V_1, \dots, F_n, V_n (GA(X) \wedge A(A_1) \wedge SF(A_1, F_1) \wedge SV(F_1, V_1) \wedge AV(X, A_1, \text{Activation-name}_1, F_1, V_1) \dots A(A_n) \wedge SF(A_n, F_n) \wedge SV(F_n, V_n) \wedge AV(X, A_n, \text{Activation-name}_n, F_n, V_n) \Rightarrow PA(X))$$

State-function (A, F) means state function F needs to be performed by agent A , which is either the monitoring agent or the data-reading agent. F is a state function use to get the state information of a GUI element or the input data state. State-value (F, V) means the state value returned by state function F at automation time. Activation-value (X, A ,

Activation-name, F, V) means the captured state value for this state function F of action X . If the returned state value is the same as the captured value, then action X will be sent to the playing agent to play. “play-action (X, V)” is a pre-defined function in the reasoning agent to be called by the rule engine. It updates the belief of reasoning agent and indicates the next GUI action that needs to be played. V is a state value. If X is an input data GUI action, it should be the data obtained from the data-reading agent. Otherwise it is a null value.

State values are stored into a GUI state file at capturing time. The user can modify these values at a later time. For example, at capturing time, we want to delete all the records with the name “John”, then if we want use this rule to delete the records with the name “Kim”, we can modify the state value “John” to “Kim” without touching the rules.

If a GUI action is an input data GUI action, then the input data is extracted and stored in an input data file at capturing time.

At automation time, facts State-Value (F, V) and Activation-value ($X, A, Activation-name, F, V$) are asserted by the reasoning agent when state values are received from monitoring agent or input data reading agent. The input data agent uses a “next_data” state function to get the next input data in an input form. It may return either a null value or an input value. If it is not a null value, the data-reading agent will modify the value V of Activation-value ($X, A, Activation-name, F, V$) that is found in the message that is received from the reasoning agent to the value V in State-Value (F, V), which is also found in the message. This makes an input data field always true for its rule. It enables the user to change the input data later without affecting this rule. The new value is then sent back to the reasoning agent. Thus, this rule is activated and the input data GUI action is sent to the playing agent.

During the automation time the playing agent or the data reading agent use the state functions to capture states. So the playing agent and data reading agent must provides the same implementation of state functions as provided by the capturing application.

The monitoring rule generated for state actions is as following:

IF: there are state actions for the GUI action.

Chapter 4 Automatic Operating Model for Product Applications

THEN: send state functions to the monitoring agent or input data reading agent.

The expression of the monitoring rule is as following:

$$GA(X) = \text{GUI-action}(X)$$

$$AG(A) = \text{Agent}(A)$$

$$SF(A, F) = \text{State-function}(A, F)$$

$$PA(F, P) = \text{Parameter}(F, P)$$

$$SM(X) = \text{Start-monitor}(X)$$

$$SA(X, A, F, P) = \text{State-action}(X, A, F, P)$$

$$SS(X, A, F, P) = \text{Send-state}(X, A, F, P)$$

$$\forall X, A, F, P (GA(X) \wedge AG(A) \wedge SF(A, F) \wedge PA(F, P) \wedge SM(X) \wedge SA(X, A, F, P) \Rightarrow SS(X, A, F, P))$$

The reasoning agent triggers this rule before sending a GUI action to the playing agent. The reasoning agent asserts a “Start-monitor (X)” fact into the rule engine to trigger a monitoring rule.

“Send-state” is a pre-defined function in reasoning agent, which is called from the rule engine and it modifies the reasoning agent’s belief base.

For example, the monitoring rule for rule “if Action *B* is complete and it is from ‘sg@jobstreet.com’ then executes action *C*” is represented as below:

$$X = \text{action } C$$

$$A = \text{the playing agent}$$

$$F = \text{get_text}$$

$$P = \text{From}(0)$$

$$GA(X) \wedge AG(A) \wedge SF(A, F) \wedge PA(F, P) \wedge SM(X) \wedge SA(X, A, F, P) \Rightarrow SS(X, A, F, P)$$

Chapter 4 Automatic Operating Model for Product Applications

The action rule for rule “if Action B is complete and it is from ‘sg@jobstreet.com’ then executes action C ” is represented as below:

$X = \text{action } C$

$A_1 = \text{the monitoring agent}$

$A_2 = \text{the monitoring agent}$

$F_1 = \text{get_text}$

$F_2 = \text{delay}$

$$GA(X) \wedge A(A_1) \wedge SF(A_1, F_1) \wedge SV(F_1, V_1) \wedge AV(X, A_1, \text{Activation-name}_1, F_1, V_1) \wedge A(A_2) \wedge SF(A_2, F_2) \wedge SV(A_2, V_2) \wedge AV(X, A_2, \text{Activation-name}_2, F_2, V_2) \Rightarrow PA(X)$$

The following expected states are generated

Activation-value (action C , the monitoring agent, Activation-name₁, delay, complete)

Activation-value (action, the monitoring agent, Activation-name₂, get_text, sg@jobstreet.com)

The user is able to modify the expected states values in the expected states file. By changing “sg@jobstreet.com” to “John”, this task could open the email from “john” and delete the email which is not from “john”.

4.5 Agents definitions

4.5.1 Monitoring agent

The monitoring agent is used to monitor the states of a GUI application.

The monitoring agent has the following beliefs:

$b_1 = \text{unexpected_state_value}(\text{functiona_name}, \text{gui_element_name}, \text{state_value}).$

It holds the unexpected state value which is captured by the monitoring application. The functiona_name is the state function provided by the monitoring agent that is used to get a property value of a GUI element. The state_value is the property value of a GUI

Chapter 4 Automatic Operating Model for Product Applications

element that indicates a GUI unexpected exception has happened. If *state_value* is null, then no unexpected exception has happened.

b₂=expected_state_value set

It holds a set of *expected_state* which are captured by the monitoring agent. An *expected_state* has three arguments:

expected_state(functiona_name, gui_element_name, state_value). The *functiona_name* is the state function provided by the monitoring agent that is used to get a property value of a GUI element. The *state_value* is the value captured by the monitoring agent.

The monitoring agent has the following goals:

g₁=monitor_unexpected_states: achieve (unexpected_state_value(functiona_name, gui_element_name, state_value) ^ state_value ≠ null)

This goal is to find an unexpected state that has happened. When the monitoring agent receives a *new_task* event from the reasoning agent, or a *resume_finished* event from the *restore_context_plan*, plan *monitor_unexpected_states_plan* will be initiated to achieve this goal. This goal is achieved when an unexpected state value is captured.

g₂=get_expected_states

It is a perform goal. A **perform goal** specifies the agent desires to execute some behavior. This goal is to get the GUI state value. When the monitoring agent receives an *expected_states* event from the reasoning agent, plan *get_expected_states_plan* is initiated.

g₃=restore_context

It is a perform goal. It restores the saved context. When the monitoring agent receives a *resume_context* event from the reasoning agent, plan *restore_context_plan* will be initiated to perform this goal. After the plan is finished, it generates a *resume_finished* internal event to start goal *monitor_unexpected_states*.

The monitoring agent has the following plans:

Chapter 4 Automatic Operating Model for Product Applications

Plan *monitor_unexpected_states_plan***Invocation Condition**

On receiving event *new_task* or event *resume_finished*

Body

task_name $\leftarrow$ *extract_task_name* (event *new_task* or event *resume_finished*)

unexpected_states_to_be_monitored_list $\leftarrow$ *load_unexpected_states* (*task_name*)

while (*b₁* is null)

wait (interval)

foreach *state_function* **in** *unexpected_states_to_be_monitored_list*

state $\leftarrow$ *execute_state_function* (*state_function*)

if (*state* = any states in *unexpected_states_to_be_monitored_list*)

then

b₁ $\leftarrow$ *state*

break

end if

loop

loop

save_context ()

Failed

inform (reasoning agent, “failure”)

Succeed

inform (reasoning agent, *unexpected_state_value*)

inform (playing agent, *unexpected_state_value*)

Chapter 4 Automatic Operating Model for Product Applications

inform (data reading agent, *unexpected_state_value*)

When the monitoring agent receives a message from the interface agent, it indicates that the user has started a new task. It first reads all the unexpected states functions from this task's unexpected states file into the *unexpected_states_to_be_monitored_list*. Then it begins to check whether these states have happened at every predefined period time. If an unexpected state has happened, it first pushes all the context information of the monitoring agent into a stack and then it informs the reasoning agent, playing agent and data reading agent. The context information includes the expected and unexpected states.

Plan *get_expected_states_plan*

Invocation Condition

On receiving event *new_state_value*

Body

state_functions_list $\leftarrow$ extract_functions (*new_state_value*)

execute_state_function (delay, delay_time)

$b_2 \leftarrow \{ \}$

foreach state_function **in** state_functions_list

if state_function $\neq$ "delay" **then**

expected_state_value_list $\leftarrow$ execute_state_function (state_function)

$b_2 \leftarrow b_2 + \{value\}$

end if

loop

Failed

inform (reasoning agent, "failure")

Succeed

Chapter 4 Automatic Operating Model for Product Applications

inform (reasoning agent, b_2)

When the monitoring agent receives a message that asks for states values from the reasoning agent, it first executes the “delay” state function. After the time in the parameter of “delay” state function has elapsed, it begins to execute other state functions to get the GUI state values. After all the state values have been obtained, it sends a message containing the state values to the reasoning agent.

Plan *restore_context_plan*

Invocation Condition

Has received event *resume_context*

Body

restore_context()

inform (reasoning agent, *expected_state_value_list*)

create_event(*resume_finished*)

Failed

inform (reasoning agent, “failure”)

Succeed

dispatch_event (*resume_finished*)

When a message is received from the reasoning agent indicating that the unexpected states has been processed, it pops up the saved context information from its stack, and replaces the expected state and unexpected state with the saved states. It then sends the states values to the reasoning agent. Thus the interrupted process is resumed.

4.5.2 Playing agent

The playing agent is used to execute GUI actions.

The playing agent has the following beliefs:

Chapter 4 Automatic Operating Model for Product Applications

$b_1 = \text{gui_action}(\text{action_name}, \text{action_parameter})$: The *action_name* is a component level action. The *action_parameter* is the input data for an input field. For a mouse related action it is null.

The playing agent has the following goals.

$g_1 = \text{play_action}$

Play action is a perform goal. It performs a GUI action. When the playing agent receives a *gui_action* event from the reasoning agent, it starts *play_action_plan* to perform this goal.

$g_2 = \text{save_context}$

save_context is a perform goal. It saves its context. When the play agent receives a *unexpected_state* event from the monitoring agent, plan *save_context_plan* is started to perform this goal.

$g_3 = \text{restore_context}$

It is a perform goal. It restores the saved context. When the playing agent receives a *resume_context* event from the reasoning agent, plan *restore_context_plan* is started to perform this goal.

The playing agent has the following plans:

Plan *play_action_plan*

Invocation Condition:

On receiving *play_action* event

Body

$b_1 \leftarrow \text{get_gui_action}(\text{play_action})$

play_action (b_1)

Failed

Chapter 4 Automatic Operating Model for Product Applications

inform (reasoning agent, “failure”)

When the playing agent receives a message from the reasoning agent to play a GUI action, it translates the GUI action event into mouse and keyboard event and sends this event to the underlying windowing system.

Plan *save_context_plan*

Invocation Condition

On receiving a *save_context* event

Body

save_context()

Failed

inform (reasoning agent, “failure”)

Succeed

inform (reasoning agent, “ready”)

When the playing agent receives a message, which indicates that an unexpected state has happened, from the interface agent, it saves the context information of the playing agent into its stack. The context information includes the next action to be played.

Plan *restore_context_plan*

Invocation Condition

On receiving event *resume_context*

Body

restore_context()

Failed

inform (reasoning agent, “failure”)

Succeed

inform (reasoning agent, “ready”)

When the playing agent receives a message, which indicates the unexpected state has been processed, from the reasoning agent, it pops up the saved context from its stack and restores its context.

4.5.3 Input data reading agent

The functionality of the data reading agent is similar to the monitoring agent. It provides input data information of an input form to the reasoning agent.

The monitoring agent has the following beliefs:

$b_1 = \text{input_data}$ set. The *input_data* set holds a set of *input_data_form*. Each *input_data_form* has two arguments:

input_data_form (*form_name*, *input_field_list*). *input_field_list* is an array of *input_field*. Each field has two arguments.

input_field(*field_name*, *field_value*)

$b_2 = \text{pointer}$ (*input_data_form_name*, *input_field_name*)

It holds the next data to be read. It has two arguments.

The input data reading agent has the following goals:

$g_1 = \text{load_data}$

This is a perform goal. It loads all the input data which is related to this task. When the data reading agent receives a *new_task* event from the interface agent, or a *resume_finished* event from the *restore_context_plan*, plan *new_task_plan* is initiated to perform this goal.

$g_2 = \text{get_state_value}$:

This is a perform goal. When the data reading agent receives a *data_state* event from the reasoning agent, plan *get_data_plan* is initiated to perform this goal.

$g_3 = \text{save_context}$:

`save_context` is a perform goal. It saves its context. When the play agent receives a *unexpected_state* event from the monitoring agent, plan *save_context_plan* is initiated to perform this goal.

$g_4 = \text{restore_context}$

It is a perform goal. It restores its saved context. When the playing agent receives a *resume_context* event from the reasoning agent, plan *restore_context_plan* is initiated to perform this goal.

The data reading agent has the following plans:

Plan *new_task*

Invocation Condition

On receiving event *new_task*

Body

$\text{task_name} \leftarrow \text{extract_task_name}(\text{event } \textit{new_task} \text{ or event } \textit{resume_finished})$

$b_1 \leftarrow \text{load_unexpected_states}(\text{task_name})$

$\text{input_data_form_name} \leftarrow \text{get_form_name}(b_1)$

$\text{initialize_pointer}(\text{input_data_form_name}, \text{null})$

Failed

$\text{inform}(\text{reasoning agent}, \text{"failure"})$

Succeed

$\text{inform}(\text{reasoning agent}, \text{"ready"})$

When a message, which indicates a new task is entered by the user, is received from the interface agent, it loads all the input data of this task.

Chapter 4 Automatic Operating Model for Product Applications

Plan *get_data_plan***Invocation Condition**

On receiving event *data_state* or event *resume_finished*

Body

form_name ← *extract_form_name* (*data_state*)

state_function_name ← *extract_function_name* (*data_state*)

if *state_function_name* = “check_data” **then**

value ← *check_data* (*form_name*)

else

value ← *next_data* (*form_name*)

end if

Failed

inform (*reasoning agent*, “failure”)

Succeed

inform (*reasoning agent*, *value*)

When the data reading agent receives a message from the reasoning agent to check input data states or read input data, it checks the action name and the input form name in the message. If it is a “check_data” action, it will check whether there is any input data of a continuous input data form remained unread. If it is true, a “true” value will be sent to the reasoning agent, otherwise a “false” value is sent. If the action name is “next_data” action, it gets the next unread data in the input from. If there is a value that has not been read, this value is sent to the reasoning agent, otherwise a null value is sent.

Plan *save_context_plan***Invocation Condition**

Chapter 4 Automatic Operating Model for Product Applications

On receiving a *save_context* event

Body

save_context()

Failed

inform (reasoning agent, “failure”)

When the data reading agent receives a message, which indicates that an unexpected state has happened, from the interface agent, it saves the context information of the data reading agent into its stack. The context information includes the input data and data pointer.

Plan *restore_context_plan*

Invocation Condition

On receiving event *resume_context*

Body

restore_context()

create_event(resume_finished)

Failed

inform (reasoning agent, “failure”)

Succeed

dispatch_event(resume_finished)

When the data reading agent receives a message, which indicates the unexpected state has been processed, from the reasoning agent, it pops up the saved context from its stack and restores its context, and then activates *get_data_plan* then sends the data or data state to the reasoning agent.

4.5.4 Reasoning agent

The reasoning agent is used to execute a specific task with the help of the monitoring agent, playing agent and input data reading agent. The reasoning agent has an embedded rule engine.

The reasoning agent has the following beliefs:

$b_1 = \text{gui_action}(\text{name})$. It holds the GUI action name to be executed.

$b_2 = \text{states}$ set. It holds a set of expected states that are to be sent to the monitoring agent and data reading agent. An *expected_state* has three arguments:

$\text{expected_state}(\text{functiona_name}, \text{gui_element_name}, \text{state_value})$. The *functiona_name* is the state function name which is provided by the monitoring agent that is used to get a property value of a GUI element.

$b_3 = \text{exception_number}(\text{number})$. It is an integer that indicates the number of unexpected exceptions that have happened so far. It has an initial value of 0.

The reasoning agent has the following goals:

$g_1 = \text{process_task: achieve}(\text{gui_action}(\text{name}) \wedge (\text{name} = \text{"end"})$

This is an achieve goal. It processes a task until the end of a task. Plan *start_task_plan* is started when the reasoning agent receives a *new_task* event from the interface agent. This goal is achieved when the GUI action name is “end”.

$g_2 = \text{process_states}$

This is a perform goal. It processes a state received from the monitoring agent or input data reading agent. Plan *process_states_plan* is started when the reasoning agent has received a *state_value* event from the monitoring agent and the input data reading agent.

The reasoning agent has the following plans:

Plan *process_task_plan*

Chapter 4 Automatic Operating Model for Product Applications

Invocation Condition

On receiving event *new_task* or event *unexpected_state_value* event

Body

task_name $\leftarrow$ extract_task_name (event *new_task* or event *unexpected_state_value*)

if event = *unexpected_state_value* **then**

$b_3 \leftarrow b_3 + 1$

local_exception_number $\leftarrow b_3$

save_context()

end if

load_rules(task_name)

$b_2 \leftarrow$ assert (start-monitor (task-name))

send_states (b_2)

while $b_1 \neq$ "end"

if $b_1 =$ "resume" and local_exception_number = b_3 **then**

resume_context();

send_event (resume_context)

$b_3 \leftarrow b_3 - 1$

break;

else if local_exception_number = b_3 **then**

dispatch_event (*process_states*)

end if

loop

Failed

inform (interface agent, “failure”)

Succeed

inform (interface agent, “success”)

When the reasoning agent receives a message, which indicates a new task is entered by the user, from the interface agent, it loads the rules of this task from the rules file & facts file into its rule engine and it also loads the expected state file of this task. Then it asserts a “Start-monitor” fact that triggers a monitoring rule of the task. This model supposes that there is always a special monitoring rule, which is not attached to any actions but to the task. This gives a way to make sure that all the conditions are met before a task is started. To start a task, the reasoning agent asserts a “Start-monitor” fact with the task name as its slot value.

When the reasoning agent receives a message that indicates an unexpected exception has happened, it first pushes the context of the reasoning agent into its stack. Then it loads the rules of the unexpected exception from rules & facts file into its rule engine, and it asserts the unexpected state to trigger a rule. If the action name equals to “resume”, then it sends resume message to the monitoring agent, playing agent and data reading agent.

When an unexpected exception happens, a new instance of plan *process_task_plan* is created.

Plan *process_states_plan*

Invocation Condition

On receiving event *process_states*

Body

send_action(*b₁*)

wait_for(monitored agent, *b₁*)

wait_for(data reading agent, *b₁*)

Chapter 4 Automatic Operating Model for Product Applications

```

 $b_1 \leftarrow \text{assert}(\text{state\_values})$ 

 $\text{assert}(\text{start-monitor}(b_1))$ 

 $b_2 \leftarrow \text{assert}(\text{start-monitor}(\text{task-name}))$ 

 $\text{send\_states}(b_2)$ 

```

Failed

```

return failure

```

Succeed

```

return success

```

When the reasoning agent receives state values from the monitoring agent or data reading agent, it extracts the values received from the messages that are sent by the monitoring agent or data reading agent and asserts the state values into the rule engine. A corresponding action rule will be fired and the action name is obtained. Before sending the GUI action to the playing agent, it asserts a “Start-Monitor(X)” fact to its rule engine, where X is the action name, and triggers the GUI monitor rule of this action. This rule sends all the state functions to the monitoring agent and the data reading agent. After that, it sends the GUI action, which needs to be played, to the playing agent.

Since what action is taken to respond the specific states is captured by the capturing application and represented as rules, by asserting the state values into the rule engine this plan is able to get the action to be played by the playing agent. That means the playing agent could play different actions based on the execution context as long as the knowledge exists.

4.5.5 Interface agent

Interface agent is used for the user to start a new task and receive executing result information of a task.

The interface agent has the following goals:

$g_1 = \text{new_task}$

This is a perform goal. It delegates the task to the reasoning agent. When the user enters a task name from the interface agent's GUI, plan *new_task_plan* is initiated to perform this goal

$g_2 = \text{display_message}$

This is a perform goal. It displays a message on the GUI. When a *result_message* event is received from other agent, plan *display_message_plan* is initiated.

The interface has the following plans:

Plan *new_task_plan*

Invocation Condition

On receiving an event *new_task* event from the GUI of interface agent

Body

send_event (reasoning agent, *new_task*)

send_event (monitoring agent, *new_task*)

send_event (input data reading agent, *new_task*)

Failed

return failure

Succeed

return success

The interface agent has a GUI for the user to enter task names. When the user enters a task name, a *new_task* event is generated.

Plan *display_message_plan*

Invocation Condition

Chapter 4 Automatic Operating Model for Product Applications

On receiving an event *result_message* from other agents

Body

message $\leftarrow$ extract_message (result_message)

display_message(message)

Failed

return failure

Succeed

return success

This plan displays message on the GUI of the interface agent.

4.6 Agents interactions

Figure 4-9 and Figure 4-10 show two scenarios for interactions between agents.

In Figure 4-9, the user starts a task through the interface agent. The task name is sent to the reasoning agent, monitoring agent and data reading agent. The reasoning agent loads the rules of this task into its rule engine. The monitoring agent loads all the unexpected states of the task. The data-reading agent loads all the input data of this task. The reasoning agent asserts a “Start-monitor (X)” fact into its rule engine, where X is the task name. Then a monitoring rule is triggered and state functions are retrieved. The state functions are sent to the monitoring agent or the data reading agent. State values are returned to the reasoning agent from the monitoring agent or input data reading agent. These states are asserted to the rule engine and the next GUI action to be played is obtained. Before sending the GUI action to the playing agent, the reasoning agent asserts a “Start-monitor (X)” fact into its rule engine, where X is the action name. Then a monitoring rule is triggered and all state functions of this action are retrieved. The state functions are sent to the monitoring agent or the data reading agent. After that, the GUI action, which needs to be played, is sent to the playing agent. When the name of the GUI action is “end”, a finish message is sent to the interface agent.

Chapter 4 Automatic Operating Model for Product Applications

Figure 4-10 shows that an unexpected state happens when a task is executing. When the monitoring agent detects an unexpected exception happens during the execution of a task, it first saves its execution context and notify the reasoning agent, playing agent and data reading agent. The three agents save their execution context. Then the reasoning agent will assert the unexpected state into its rule engine. An action rule for this unexpected state is fired. The remained processes are the same as a normal task processing. When an action name “resume” is met, the reasoning agent first resumes its saved context and sends a message to the monitoring agent, the playing agent and the data reading agent. The three agents restore its saved context and ready for the execution of interrupted task. The monitoring agent and data reading agent resume the original task process by sending the current states value.

Chapter 4 Automatic Operating Model for Product Applications

Figure 4-9 GUI automation agents' interaction scenario for expected states

Chapter 4 Automatic Operating Model for Product Applications

of a task using the capturing application. The knowledge about the task is captured in the form of rules and facts. Input data and state values are extracted from rules and stored into separated text files. It is easy for the user to modify later.

The capturing application is also able to record the user's actions when the user handles unexpected states. The process of an unexpected state is treated as a task as well.

Multiple agents work together to achieve a GUI task. The model is able to play different actions based on the states gathered by the data reading agent and monitoring agent at run time.

When an unexpected exception happens during the execution of a task, the related agents could save their execution context into stacks and begin to deal with the exception. After the exception has been processed, the normal execution of the task is resumed.

Chapter 5 Automatic Operating Model for GUI Regression Testing

The characteristics of the GUI applications under test are different from that of the released product GUI applications. One difference is that the GUI applications under test are changed more often than the released GUI applications. Another difference is that the replaying tool needs to stop immediately if there is a mismatch between the GUI state and predefined baseline. In this chapter, we propose a multi-agent based model to facilitate and automate the regression testing of GUI applications. A new actionable knowledge model to represent test cases for a GUI application is also proposed. This makes the test case repair easier. The organization of this chapter is as follows: Section 5.1 is the overview of the research. Section 5.2 is the algorithm to convert the captured test cases into actionable knowledge model. An algorithm for reconstructing a test case is presented in Section 5.3. Test case repair and affected test cases finding algorithm are discussed in Section 5.4. Section 5.5 is the system architecture. Section 5.6 describes the interactions between agents. Section 5.7 is the summary.

5.1 Overview

As we discussed in Chapter 2, Modern GUI applications may be composed from a number of different software components. For example, a GUI application may access remote databases, or other machines; or run as a standalone application. We could categorize GUI applications as monolithic application, client/Server application, and distributed application.

To ensure the correctness of GUI software, comprehensive testing is needed. Testing GUIs is more complex than testing conventional software. Not only does the underlying software have to be tested, but the GUI itself also must be exercised and tested to check whether it conforms to the GUI specifications [3].

GUI software has the following characteristics as compared to the traditional ones:

- A GUI application is an event-driven application. The user could perform actions in an unpredictable sequence.
- The expected output of a GUI application is difficult to be predefined in the test case for comparison.
- The partitioning of computing in client-server or distributed GUI applications makes the creation and execution of test cases become complex.

GUI testing requires test cases (sequence of GUI events that exercise GUI widgets) to be generated and executed on the GUI. However, the current available techniques for obtaining GUI test cases are resource intensive and require significant human intervention. Even though a few automated GUI test case generation methods have been proposed, in practice, test cases are still being generated manually using capture/replay tools [1].

These testing tools capture all user interactions with the application and store these actions into a script file [1]. These actions include keystrokes, mouse movements, menu selection, and button clicks. Testing tools also recognize and capture responses that the user sees, such as windows changing, drop-down list boxes appearing, messages and dialog windows displaying and so on. When the testing tool replays the recorded script at a later time, it initiates the same action that the original user performed. The test tool also compares responses that occur during the playback testing phase with those that occurred during the original recording cycle. The test programs then log the difference [70]. At any time the tester can stop the recording and insert other actions such as verification to a database table, verification to a GUI element, etc. to the test case.

Using “capture/replay” method to create test cases is very time consuming. Any errors occurred during recording time will cause re-recording of the test cases. As different test cases are stored in different script files, any modification to the GUI application, such as a GUI element is added or deleted from the GUI application, will cause one or more test cases that are related to the GUI element become invalid and discarded. These test cases need to be re-recorded.

Chapter 5 Automatic Operating Model for GUI Regression Testing

A test case repair algorithm is proposed in [1]. This algorithm is automatically trying to repair test cases that are generated manually by a “capture/replay” tool. It uses two kinds of graphs to represent a GUI application. One is GUI control-flow graph (G-CFG) and the other one is the GUI call graph (G-call graph). A G-CFG statically represents all possible interactions among the events in a component. A G-call graph shows the invoke relationships among all the components in the GUI. These graphs are automatically obtained from the GUI by performing a traversal of the GUI’s event structure. Then the regression tester takes them as input to the G-CFGs and G-call tree for both the original and modified GUI. A test case is classified into unusable and usable test cases. For a repairable test case, the algorithm skips events or tries to insert a single new event until a legal event sequence is obtained.

This algorithm is suitable for the GUI applications which the input data fields have no constraints, for example, a notepad application. We can enter anything in the notepad’s text field.

But to automatically repair a test case generated manually by using a “capture/replay” tool, the GUI structure of a GUI application must be obtained first. As we know, not all GUI applications are like notepad. Most of the GUI applications have the underlying constraints to the GUI element. Navigation to most of the GUI applications relies on the input data from keyboard. These input data must follow some built-in constraints, such as email address, identity card number, account number etc... If the correct input data can not be generated automatically, using these tools to obtain the GUI structure automatically will fail. In such a case, the tester needs to generate it manually. Even the GUI structure can be obtained, when repairing the test case automatically using the above algorithm, if the only one event that the algorithm tries to insert is a keyboard input event and has bound constraints, then all the event sequences generated will be invalid sequences.

To facilitate test cases recording, playing and repair, we propose a multi-agent based model for GUI regression testing. In our model, test cases are represented in a sharable knowledge format instead of script files.

The use of multi-agent technology provides a flexible and scalable architecture for testing standalone GUI applications as well as distributed GUI applications.

A GUI application may have a set of test cases. A test case is defined as following [1]:

GUI test case T is a pair $(S_0, a_1; a_2; \dots; a_n)$, consisting of state $S_0 \in S_t$, which is the initial state for T , and an action sequence $a_1; a_2; \dots; a_n$ performed by the tester during recording time.

A **GUI action** is a mouse event or keyboard inputs performed on a GUI element. For example, mouse clicks on a button and a text string in a text box.

We can consider the process to capture the user's action into script files by using a "capture/replay" tool as reconstruction of the knowledge about a GUI application. The capture tool does not only capture the action that could be performed on a GUI element, but also the relationships between GUI actions. Using "capture/replay" approach, each test case is stored in a script file. This makes the knowledge about a GUI application not shareable across test cases. Thus, if the GUI is modified (GUI elements are added, deleted and renamed), we need to modify all the script files that contain the modified GUI action. For a tester who does not have programming skills, all the test cases will need to be rerecorded.

In this thesis, we divide user actions of a GUI test case into two categories: GUI related actions and non-GUI related actions. GUI related actions are used to navigate a GUI application or verify states of a GUI component. Non-GUI related actions are all the other actions in the test case besides GUI related actions, such as initialization action of a test case, database states verification actions, etc.

When a test case is being created, the tester traverses the GUI application by clicking the mouse and inputting data using the keyboard. The "capture/replay" tool is able to represent a mouse and a keyboard event at GUI component level. For example, when clicking on the "File" menu item, it is able to generate a string of "FILE.Click" to represent the tester's action.

Actions sequence $a_1; a_2; \dots; a_n$ are ordered by the time they are performed during recording time. If no errors happened during the execution of a test case, a_1 is always the first action, a_n is the last action and a_i is performed immediately after a_{i+1} .

In this thesis, all the test cases of a GUI application are integrated and represented in a proposed **Actionable Knowledge Model**. This knowledgeable model includes two parts: **Actionable Knowledge Graph (AKG)** and **Sub-Goals-List (SGL)**.

An **AKG** for a GUI application is 2-tuple $\langle V, E \rangle$ where:

1. V is a set of vertices representing all the actions in all the test cases of a GUI application.
2. $E \subseteq V \times V$ is a set of directed edges between vertices. An edge $(v_x, v_y) \in E$ if the action represented by v_y follows the action represented by v_x at the time the tester creating test cases.

As all test cases of a GUI application are integrated into one AKG, test cases are able to share knowledge in terms of action information and relationship between two actions. For example, if several test cases have the same action flow $v_x \rightarrow v_y$, then only two vertices, v_x and v_y , and one edge $\langle v_x, v_y \rangle$ are needed for these test cases. At playing time, when a test case is to be executed, it is reconstructed from the AKG.

The AKG is able to memorize actions information and relationship between two actions only. The ordering information of actions in a test case is missed out as more test cases are added to the AKG.

For example, Figure 5-1 shows an AKG of a GUI application, which includes two test cases, T1 and T2. T1 has an action flow, $A \rightarrow C \rightarrow D$, and T2 has an action flow, $B \rightarrow C \rightarrow E$. Using the information provided by the AKG shown in Figure 5-1, we are not able to reconstruct test case T1 and T2. We may get wrong result. For example, T1 becomes $A \rightarrow C \rightarrow E$ and T2 becomes $B \rightarrow C \rightarrow D$. The problem is caused by vertex C, where two test cases are met.

Figure 5-1 AKG example

To reconstruct test cases correctly with an AKG, further information is needed besides the AKG. Our method is to use one or more goal vertices to represent a test case. By default, a test case is represented by the last vertex as its goal vertex, which represents the last action of a test case. If a later added test case shares a vertex with a former added test case, then the newly added test case uses its last vertex and this shared vertex as its goal vertices. For example, if T1 is created first, when T2 is being added to the AKG; and if action C is the action being performed by the tester, because C already exists in the application's AKG, then we use C as a goal vertex of test case T2. After T2 has been added to Figure 5-1, T2 is represented using goal vertices C and E. As T1 is created before T2, it is represented using goal vertex D. These goal vertices are to be remembered when adding a new test case to the AKG. To differentiate vertex C in T1 and T2, we give an alias name C1 to represent vertex C in test case T2 and remember the C1's preceding vertex name B as well. Before describing how to reconstruct a test case, we give the following definitions:

A **goal-vertex** is a vertex in an application's AKG, which in-degree is greater than one.

Vertex C in Figure 5-1 is a goal-vertex.

A **sub-goal** is one of **goal-vertex**'s alias names, which has a suffix identified by the in-degree of the **goal-vertex** at the time it is created.

For example, when vertex C is added to the AKG shown in Figure 5-1 and because C exists in the AKG already, an alias name C1 is created for action C in test case T2. The alias name C1 is composed by the action name C with the in-degree 1 of vertex C as its suffix. Then C1 is a sub-goal.

Exception: a vertex, which represents the last action of a test case, is always a sub-goal even it has no alias name.

A SGL consists of a list of sub-goals in the order, which they are created.

In this thesis, **all** the test cases of a GUI application are represented as **AKG**.

When reconstructing a test case represented in AKG, each sub-goal in a SGL is used to create one test case segment. All the test case segments are combined together in the same order as they are in the SGL.

As the sub-goal is only the vertex which in-degree is greater than one in the AKG of a GUI application, the new representation is shorter than the original test case action sequence. This implies a higher level representation of a test case. It makes changes that made to one part of the GUI, which does not invalidate the other test cases.

When a GUI application is modified, one or more test cases may become invalid. If such a test case is executed, an unexpected result will occur. As the test cases in this thesis are represented in the form of AKG, the actions information and relationship between the actions of these test cases are shared in the AKG. This makes it is possible to repair one test case and make all the other invalid test cases become valid.

Since test cases could not be repaired automatically all the time, we use an incremental method in this thesis. The tester first gives an action name of the GUI application, which is near the modified GUI area or it is the deleted action name. The system will find all the test cases that include this action and list them to the tester. The tester can select any one of these listed test cases to do the repair work. The system will automatically execute the GUI application to the specified GUI action or the previous action if the action is deleted. The tester then manually re-creates the partial test case in this area. During the repair session, if the system finds that the tester is creating an existing edge in the test case, it will prompt the tester whether the repair is finished. The tester can terminate the repair at any time. During the repair of one test case, the system will remember test cases affected by the current repaired test case. If the repair for the selected test case is finished, the system will help the tester to repair other affected test

cases, which are either caused by the modification of the GUI application or caused by the current repair, until the tester decide to terminate the repair.

For example, in Figure 5-2, there are three test cases T1, T2 and T3. T1's action flow is $A \rightarrow C \rightarrow D \rightarrow E \rightarrow F$. T2's action flow is $B \rightarrow C \rightarrow D \rightarrow E \rightarrow F$ and T3's action flows is $F \rightarrow G$. If action D is deleted from the GUI application and C is an action that is near the deleted action D, the tester first tells the system that D is a deleted action. The system will detect that test case T1 and T2 are affected. If the tester selects T1 to do the repair, the system will execute test case T1 to C automatically. Then the tester does the repair manually. The tester then performs action E and then action F. As edge $\langle E, F \rangle$ already exists in test case T1, the system will ask the tester whether the repair is finished. If the tester answers "yes", the system will stop the repair process. Otherwise, the system will continue to carry on the repair. A new edge $\langle C, E \rangle$ is added to the system after the repair is done. In this case, test case T2 also becomes valid after the repair of T1.

Figure 5-2 AKG example for test case repair automatically

In Figure 5-3, if C is deleted, the tester may use either T1 or T2 to begin the repair process. During the repair session of test case T1, the system will execute the GUI application to A automatically. The tester then performs action D and E. and decides to stop the repair at this point. After T1 has been repaired, vertex C, edge $\langle A, C \rangle$, edge $\langle C, D \rangle$ and edge $\langle B, C \rangle$ are deleted from the system with the agreement of the tester. A new edge $\langle A, D \rangle$ is then created. If the tester wants the system to modify T2 automatically, which means using D to replace C, then the system will add a new edge $\langle B, D \rangle$ to the system and modify test case T2's SGL automatically. In this case, T2 is also repaired automatically. If the tester, instead of adding edge $\langle B, D \rangle$ automatically,

wants to modify test case T2 by linking actions B and E in the test case T2, the system will help the tester to repair test case T2 and execute the GUI application to vertex B automatically after T1 gets repaired. The tester then performs action E and F and decides to stop the repair at this point. In both cases, T2's SGL needs to be modified, because vertex D and E's in-degree becomes greater than 1.

Figure 5-3 AKG example for test case repair manually

In a distributed environment, the actions in a test case are executed in different locations. When creating a new test case, the tester needs to operate both local and remote GUI applications. All the actions performed by the tester are captured and combined into one test case. When replaying the test case, each action in a test case is sent to the corresponding location to execute and the execution results are stored in one common place.

A Software agent is a computer program, which performs tasks on behalf of another entity, possibly over an extended period of time without direct supervision or control.

A multi-agent system is a construction of complex systems involving agents. It involves mechanisms that allow agents to act through behaviors and to coordinate these activities. These agents work independently but also together to achieve some common goals. To help the tester to perform the GUI testing efficiently, a multi-agent system model is proposed in this thesis.

Chapter 5 Automatic Operating Model for GUI Regression Testing

In the chapter we present a multi-agent based model to do GUI regression testing. In this model, multiple agents are defined. An interface agent is defined for the user to interact with the agent system. At the recording time, agents work together to convert test cases of a GUI application into knowledge format of AKG. Later if the GUI is changed, with the help of the tester and the agents, the knowledge representation could be updated. At the playing time, the agents could work together to reconstruct a test case from the knowledge representation and execute it.

The contribution of this chapter includes:

- 1) A knowledge based representation of all the test cases of a GUI application. This representation makes it is possible to repair one test case that causes some of other test cases get repaired as well if the modified GUI element is shared by more test cases. This representation also enables the tester reuse actions in the existing test cases to create a new test case, if the new test case is executing actions that have been recorded before.
- 2) Several algorithms that are used to operate the knowledge model.
- 3) A multi-agent based model to help the tester to create, repair and execute test cases. The model is suitable to test distributed GUI applications.

5.2 Test case conversion algorithm

This algorithm processes one action each time when it is called. It takes five parameters, i.e. test case name being generated, current action name, previous action name, the AKG and the sequence number, which is 0 when a first new test case is being created, to identify the parameters. The knowledge model is updated when the algorithm is called.

This algorithm is used by the KB managing agent to represent the test cases using the knowledge model at capturing time of the test case.

ALGORITHM : convert GUI action to knowledge

DATA: TS: test case name,

Chapter 5 Automatic Operating Model for GUI Regression Testing

CA: current action,

PA: previous action,

g (V: vertices, E: edges): Graph,

SEQ: number to indicate the order of parameter

sub_goals: sub goals list

RESULT: updated g and sub_goals

newEdge←false;

BEGIN

IF (CA=="FINISHED") **THEN** // if a test case is finished

process_last_action(PA,g,sub_goals)

ELSE

IF (PA==null) **THEN** // if the action is the first action of a test case

process_first_action(CA,PA,g, SEQ, sub_goals)

ELSE // it is not the first action in a test case

process_mid_actions(CA,PA,g, SEQ, sub_goals)

END IF

PA←CA //save this action as previous action

SEQ←SEQ+1 //increase the order number

END

Procedure *process_last_action* adds the last action to the SGL.

PROCEDURE process_last_action

DATA: TS: test case name,

Chapter 5 Automatic Operating Model for GUI Regression Testing

PA: previous action,

g (V: vertices, E: edges): Graph,

sub_goals : sub goals list

RESULT modified sub_goals

BEGIN

inDegg←inDegreeOf(g, PA); // get the in-degree of the action

IF (inDeg==1 || newEdge==false) **THEN** // add the last action as a sub-goal of a test case

sub_goals←sub_goals + { PA }

PA←null // ready to process a new test case

END

Procedure *process_first_action* processes the first action in the test case.

PROCEDURE process_first_action

DATA: TS: test case name,

CA: current action,

PA: previous action,

g (V: vertices, E: edges): Graph,

SEQ: number to indicate the order of parameter

sub_goals: sub goals list

RESULT: updated g and sub_goals

BEGIN

sub_goals←{} //initialize the sub goals set of a test case

V←V+ {TS} // add the test case name as the first vertex

Chapter 5 Automatic Operating Model for GUI Regression Testing

```

save "parameter(TS, TS, "",SEQ) // save the parameters

IF (CA  $\in$  V) THEN // the action already exists

    in-degree $\leftarrow$ inDegreeOf(CA); // get the in-degree of the action

    alias-name $\leftarrow$ CA +in-degree //create the alias name

    save "goal_alias(alias-name,CA)" //save alias

    sub_goals $\leftarrow$ sub_goals + { alias-name} // add it as a sub-goal

    E $\leftarrow$ E + {<TS,CA> }; V $\leftarrow$ V + {CA} //update the graph

    Para $\leftarrow$ paramter of the CA

    save "parameter(TS, alias-name,Para, SEQ) //save parameters

    save "precedes(TS, alias-name)" //save previous action

    // if this action has a alias name

ELSE // a new action

    V $\leftarrow$ V + {CA}

    E $\leftarrow$ E + {<TS,CA> } //update the graph

    Para $\leftarrow$ paramter of the CA

    save "parameter(TS, CA, Para,SEQ) //save parameters

END IF

END

```

Procedure *process_mid_actions* processes the actions in a test case except the first action.

PROCEDURE process_mid_actions

```

IF (CA  $\notin$  V) THEN // the action is a new action

```


Chapter 5 Automatic Operating Model for GUI Regression Testing

```

newEdge←true;

V←V + {CA} ;E←E + {<PA,CA> } //update the graph

save “parameter(TS, CA, Para,SEQ) //save paramters

ELSE // the action is performed before,

IF (<PA,CA> ∉ E) THEN // no path from previous action to this action

    newEdge←true;

    in-degree←inDegreeOf(CA);

    alias-name←A + in-degree //new alias name

    E←E + {<PA,CA> } //update the graph

    save “precedes(PA, alias-name)” //save its previous action

    // if this action has a alias name

    save “goal_alias(alias-name,CA)”

    sub_goals←sub_goals + { alias-name} //save sub-goal

    save “parameter(TS, alias-name,Para, SEQ) //save parameters

ELSE// path already executed before

    newEdge←false;

    save “parameter(TS,CA, Para,SEQ) //only save parameters

END IF

END IF

END

```

We use four facts to represent other knowledge of the test cases of a GUI application. Fact “precedes (a1, a2)” represents the preceding action of a sub-goal. Fact “parameter (test-case-name, action-name, parameters, order-info)” represents the parameters of

each action. Fact “goal_alias(alias-name, action-name)” remembers the vertex name of a sub-goal.

In the following part of this section, we give an example to demonstrate how this algorithm works.

Suppose we have a GUI application shown in Figure 5-4. It is used to add staff information. To add a new staff to the system, the tester first click “New” button, and fill an input form with the staff’s name, age and click “Save” button. The tester clicks “Exit” button to quit the application. The following action sequence is the test case to add two new different staffs to the system.

Figure 5-4 Example GUI for regression testing

1. Setup initial states (start the application).
2. app1@location.Add New Staff.New.Click.
3. app1@location.Add New Staff.Name.Input “John”.
4. app1@location.Add New Staff.Age.Input “20”.
5. app1@location.Add New Staff.Save.Click.
6. app1@location.Add New Staff.VerifyDatabase “Staff”, “name:John,age:21”.
7. app1@location.Add New Staff.New.Click.

Chapter 5 Automatic Operating Model for GUI Regression Testing

8. app1@location.Add New Staff.Name.Input “Luc”.
9. app1@location.Add New Staff.Age.Input “21”.
10. app1@location.Add New Staff.Save.Click.
11. app1@location.Add New Staff.Exit.Click.

In the above test case, each action is represented as “application name @ location.window name + component name + event name + {parameters}”. Navigation actions have empty parameters. “VerifyDatabase” is a verification action provided by the recording agent to check if the database state is consistent with the input information. The AKG for this test case is created by the converting algorithm is shown in Figure 5-5. The meaning of each vertex is as following:

T1= app1@location.T1.Setup initial states

A= app1@location.Add New Staff.New.Click

B= app1@location.Add New Staff.Name.Input “John”

C= app1@location.Add New Staff.Age.Input “20”

D= app1@location.Add New Staff.Save.Click

E= app1@location.Add New Staff.VerifyDatabase “Staff”, “name: John, age: 21”

F= app1@location.Add New Staff.Exit.Click

Figure 5-5 AKG for test case T1

The execution result of the algorithm for each action in T1 is shown as below:

Chapter 5 Automatic Operating Model for GUI Regression Testing

1. parameter ("T1","T1","",0), parameter("T1","A","",1).
2. parameter ("T1","B","John",2).
3. parameter ("T1","C","20",3).
4. parameter ("T1","D","",4).
5. parameter("T1","E","Staff;name:John,age:21",5).
6. precedes ("E","A1") , goal_alias ("A1","A"), parameter("T1","A1","",6), sub_goals={"A1"}.
7. parameter ("T1","B","Luc",7).
8. parameter("T1","C","21",8).
9. parameter("T1","D","",9).
10. parameter("T1","F","",10).
11. sub_goals={"A1","F"}.

Now we have another test case named “add_an_existing_staff” to test adding an existing staff. This test case is executed after the first test case has been executed. As shown in Figure 5-6, the GUI application under test will popup an error dialog box after we have clicked on “Save” button. The tester then clicks on “Ok” button on the dialog box to return and then click “Exit” button to exit the application.

Figure 5-6 GUI example for add existing staff

The second test case includes the following actions:

1. app1@location.T2.Setup initial states.
2. app1@location.Add New Staff.New.Click.
3. app1@location.Add New Staff.Name.Input "John".
4. app1@location.Add New Staff.Age.Input "20".
5. app1@location.Add New Staff.Save.Click.
6. app1@location.Error.Ok.Click.
7. app1@location.Add New Staff.Eixt.Click.

The AKG of the GUI application is changed to Figure 5-7. Two new vertices T2 and G are added to the AKG.

Figure 5-7 AKG for test case T1 and T2

T2=app1@location.T2.Setup initial states

G= app1@location.Error.Ok.Click

After applying the algorithm to the second test case T2, we get the following result for each action:

1. precedes ("T2", "A2"), parameter("T2", "T2", "", 0), parameter("T2", "A2", "", 1), goal_alias ("A2", "A"), sub-goals={"A2"}.
2. parameter ("T2", "B", "", 2).

Chapter 5 Automatic Operating Model for GUI Regression Testing

3. parameter ("T2", "C", "Luc", 3).
4. parameter("T2", "D", "20", 4).
5. parameter ("T2", "G", "", 5).
6. precedes ("G", "F1") , goal_alias ("F1", "F"), parameter("T2", "F1", "", 6), sub_goals= {"A2", "F1"}.

After processing, test case T1 is represented by sub-goals-list ("A1", "F") and test case T2 is represented by sub-goals-list ("A2", "F1") which is much shorter than their original one.

5.3 Test case reconstruction algorithm

When a test case is to be executed, the test case is reconstructed dynamically from the knowledge model. The system first gets the SGL of a test case. Then, each sub-goal is used to construct one segment of the test case. At last, all the segments are combined in the same order as they are in the SGL. When constructing a segment, it will first get the vertex name of the sub-goal, which is represented as an alias name. Then it gets its preceding vertex name. The system goes along the path of the segment from the last vertex until it reaches the first vertex of the segment. After having reached the first vertex of the segment of a sub-goal, the algorithm will process each vertex in the path. When processing a vertex, its action name and parameters are combined together to represent a test case action.

The reconstruction algorithm is listed as following:

ALGORITHM Test Case Reconstruction

DATA: TS: Test case name;

g (V: vertices, E: edges): Graph) //the main algorithm

RESULT: Test case

BEGIN

subGoalList ← read_Subgoal(TS); //read the sub-goals-list of a test case

Chapter 5 Automatic Operating Model for GUI Regression Testing

```

preSubGoal←null; // the first sub-goal's preceding sub-goal is null

testCaseList←{} //list to hold all the actions

FOREACH subGoal ∈ subGoalList DO //reconstruct test case for each sub-goal

    testCaseList←ProcessASubGoal(sub_goal, preSubGoal,testCaseList);

    // construct the segment for a sub-goal

    preSubGoal =GetRealName(sub_goal);

    //get the processed sub-goal's real action name

    IF (preSubGoal ==null) THEN //the sub-goal has no alias action name

        preSubGoal ←sub_goal; //the sub-goal name is a real action name

    LOOP

RETURN testCaseList

END

```

Procesure “ProcessASubGoal” is used to reconstruct the segment for a sub-goal.

PROCEDURE: ProcessASubGoal

DATA: sub_goal:sub-goal name,

preSubGoal: preceding sub-goal in the sub-goals-list,

testCaseList: all the actions of a test case

RESULT Segment test case

BEGIN

preAction←get_preceding(sub_goal) //get the preceding action name

realActionName← GetRealName(sub_goal) //get the real action name

IF (realActionName ==null) **THEN**//not a alias action name

Chapter 5 Automatic Operating Model for GUI Regression Testing

```

    realActionName ← sub_goal; //use its original name

    IF (preSubGoal == null && preAction == null ) THEN //the first action of the test case

        appendToList(testCaseList,realName); //the test case only have one action

        RETURN testCaseList;

    ELSE IF (realActionName not equal to preSubGoal ) THEN //process its preceding action

        testCaseList ← ProcessASubGoal (realActionName, preSubGoal, testCaseList);

        appendToList(testCaseList,realName);

    END IF

RETURN testCaseList; //return the result

END

```

Function “get_preceding” is used to get the real vertex name that precedes the current vertex. Because when we are creating the representation of a test case, we have saved the vertex name of different sub-goals, and function “get_preceding” will lead the algorithm to go to the correct path of the test case.

5.4 Test case repair related algorithm

A modification to a GUI application is needed in a number of situations:

- A new GUI element is added.
- A GUI element is deleted.
- A GUI element is modified.

If a new GUI element is added, the tester needs to select a test case to add the new action to the selected test case. The tester tells the system an existing action name of the GUI application. The system could find all the test cases that include this action. The system will execute the GUI application to this point automatically using one of the found test cases. Then the tester could manually repair the test case.

Chapter 5 Automatic Operating Model for GUI Regression Testing

If a GUI element is deleted from the GUI application, the tester may tell the system the name of the action. The system will automatically find the test case being affected and execute one test case, which is selected by the tester, to the preceding action of the deleted GUI element.

If a GUI element is modified, the tester tells the new action name of the modified GUI element to the system. The system just modifies the corresponding vertex and facts.

During the repair session, whenever a new vertex's in-degree becomes greater than 1, the SGL of the affected test cases need be modified as well.

As we discussed in Section 4.2, if the GUI application is modified, a set of test cases are affected. That does not mean all the test cases need to be repaired individually. In some cases, if the GUI actions are shared by some test cases, the repair to one test case will cause other test cases to get repaired as well.

The algorithm for finding all the affected test cases is as following:

ALGORITHM FindAffectedTestCase

DATA: String actionName)

RESULT: Affected test cases name

BEGIN

affectedTestCaseList ← {}; // initialize the result set

visitedAliasSet ← {}; // initialize the visited sub-goals

// find all the affected test cases

affectedTestCaseList ← findTs(actionName, affectedTestCaseList, visitedAliasSet);

RETURN affectedTestCaseList

END

PROCEDURE: findTS

DATA: actionName: input action name;

Chapter 5 Automatic Operating Model for GUI Regression Testing

affectedTestCaseList: LIST to hold the result;

visitedActionSet: SET to hold visited actions

RESULT: Affected test cases name

BEGIN

vistedAction←getActions(visitedActionSet ,nodeName); //check if a sub goal is performed before

IF (vistedAction!=null) **THEN** //if performed then return

RETURN affectedTestCaseList;

END IF

aliasSet←getActionAlias(nodeName); //get all the sub-goal name of a vertex

// as well as itself

FOREACH as ∈ aliasSet **DO**

IF (sizeof (aliasSet)>1) **THEN** //it is a sub-goal vertex

visitedActionSet←visitedActionSet + {nodeName}; //add it to the visited list

END IF

preAction←get_preceding(as); //get preceding action

IF (preAction == null) **THEN** //finished for one sub-goal

affectedTestCaseList ←affectedTestCaseList + {nodeName};

CONTINUE LOOP;

ELSE //find all the test cases that include the preceding action

affectedTestCaseList ← findTS (preAction, affectedTestCaseList,
visitedActionSet);

END IF

LOOP

RETURN affectedTestCaseList

END

Function "getActionAlias" is used to find all the alias names of an action as well as the action itself. Set "visitedActionSet" is used to remember all the visited alias name if the vertex, which in-degree is greater than 1, is processed before. It avoids infinite loop in a test case, such as T1, which includes duplicated actions.

5.5 Architecture of the proposed GUI regression testing model

The architecture of the multi-agent based model for GUI software regression testing is shown in Figure 5-8. The functionalities of each agent are briefly described as below.

User interface agent: This is a GUI agent. It is used to accept instructions from the tester and displays information generated by the mediating agent.

Capturing agent: This agent is used to capture the tester's action when the tester operates on the GUI application under test.

Figure 5-8 Architecture of multi-agent based GUI regression testing model

Playing agent: This agent is used to perform the actions in the test case. There may be one or more playing agents in the system, such as GUI playing agent that is used to play the GUI related actions and database playing agent which is used to execute database related actions.

Remote Screen capturing agent: This agent is used to capture the physical screen of a remote GUI application.

Remote controlling agent This agent is used to display the remote screen captured by the remote screen capturing agent in the tester's desktop and sends the tester's actions to the playing agent on the remote desktop.

KB managing agent: This agent is used to do the reasoning work for the mediating agent, including creating, finding and reconstructing a test case.

Mediating agent: This agent plays the role of mediator in the system. It processes the tester's command, relays information between agents.

5.6 Agent interactions

5.6.1 Test case creation

The tester starts the recording by entering the test case name through the interface agent. The recording agent begins to record all the actions performed by the tester and send them to the mediating agent. The mediating agent sends this information to the KB managing agent. The KB managing agent then converts these actions into the AKG and SGL using the conversion algorithm. The mediating agent also monitors the user's actions history to see whether the tester is performing actions that have been recorded before. If it is true, it will send a series of its following actions to the tester, and the tester can select the actions, change the actions' parameters and execute them automatically. Paper [69] proposed an algorithm to learn the user habits. In our model, the user's actions are recorded as they are stored as test cases, so it is enough for us to check the tester's two or three latest actions and compare them with the existing test case's actions.

Chapter 5 Automatic Operating Model for GUI Regression Testing

In the case when a test case has more GUI applications involved. Then at any time, the tester could operate the remote GUI application through the remote screening capturing agent, the remote playing agent and the remote capturing agent. Currently, there are some research focuses on the remote desktops access. In our model, we focus on the functionality and have no intention to do research in this area. The remote capturing agent will send the screen images to the local remote controlling agent. The local remote controlling agent will display the screen image captured on the tester's screen. The tester then operates on the screen. The remote controlling agent will capture the tester's mouse and keyboard events and send them to the remote playing agent. The remote playing agent then plays these mouse and keyboard events at the remote desktop. The remote capturing agent will then capture these events in component level rather than low level mouse and keyboard events. The remote capturing agent sends these captured component level GUI events to the mediating agent, and the mediating agent will add these actions to the KB managing agent. The KB managing agent will then add them into AKG and SGL.

The tester can also stop the recording and insert verification actions to the test case. The KB manger agent will insert these actions into the test case.

5.6.2 Test case execution

The tester can select a test case to execute through the interface agent. The mediating agent will ask the KB managing agent to reconstruct the test case using the test case reconstruction algorithm and perform the initialization action itself. The initialization work includes the starting of all of the playing agents. Then the mediating agent sends one action to a corresponding playing agent for execution. If the action is executed successfully, then another action is sent. All the result of a verification actions are recorded into the log files by the mediating agent.

5.6.3 Test case repair

The tester gives a GUI action name to the mediating agent through the interface agent. The mediating agent asks the KB managing agent to search all the test cases that include this action using the test case finding algorithm. The names of these test cases are sent to the interface agent. The mediating agent will execute a test case selected by

the tester. The mediating agent first executes the initialization action of the test case and executes the test case to the point with the help of the playing agents. Then the recording agent is started by the mediating agent and begins to record the tester's action. The mediating agent will save all the changes to the test case. If the mediating agent detect that the tester has finished the repair task, it will ask the tester whether to stop the repair. It's up to tester to decide when to exit.

After the repair to a test case is finished, the mediating agent will list all the test case names that are affected by the current repair and send them to the interface agent. The tester will decide whether to do further repair for these test cases. The KB manger agent will do the KB related operations such as test case converting, reconstructing and finding.

5.7 Summary

In this chapter, we propose a multi-agent based model to do GUI application regression testing, which is suitable to be applied in both standalone and distributed environment.

To make a test case more adaptable to the change of the GUI of a GUI application, we define an actionable knowledge model to represent a test case. Some algorithms to operate the actionable knowledge model are also proposed.

In this proposed model, a test case with one or more GUI application distributed in different locations could be created, replayed and repaired. Our repair approach is a semi-automated way. The repair approach gives the tester a more flexible way and saves the tester's effort as much as possible.

Chapter 6 Prototype Implementation

This chapter presents two prototype systems, which are implemented to verify the models discussed in Chapter 4 and Chapter 5. In this chapter, the two prototype systems are further discussed. Section 6.1 introduces the GUI automation prototype and Section 6.2 introduces the GUI repression testing prototype. Section 6.3 is the summary.

6.1 Prototype implementation of GUI automation model

The main purpose of this prototype is to verify the model. The prototype consists of one application and five agents.

The prototype has the following feature:

- The user's actions and GUI states are manually entered by the user instead of recording them automatically, because the GUI capturing and playing related technology are mature already but very complex to implement from scratch. How to convert a mouse or keyboard action to a component level action is beyond the scope of the research.
- Java AWT events are used to simulate the events between agents and inside an agent. BDI agents communicate through events. Java AWT events are suitable to be used as the event dispatching mechanism.
- Each agent is implemented using a Java thread, and each plan of an agent is also implemented using threads. Most of the agent platform such as JADE and JACK also implements agents as threads.

Jess is used as the inference engine.

6.1.1 Capturing application implementation

Figure 6-1 is the screenshot of the GUI of the capturing application. This GUI consists of three parts. The top part is used to define a task of an application. The middle part is used to enter the GUI states functions. The bottom part is used to input user actions.

Before recording, the user first gives the task name and the application name if the task belongs to a GUI application. The processes of unexpected exceptions are also defined as tasks. The “Exception” checkbox needs to be checked when this task is to process an unexpected exception. The “Append” button is used to add a new expected state to an action. After the definition is finished, click the “Set” button, the capturing process begins.

Figure 6-1 Screenshot of capturing application

In the middle part, the user could select an agent and a state function of the agent which is predefined in the agent to get state information. The user is also able to key in new state functions. There are two predefined monitoring agents. One is GUI monitoring agent and the other one is the input data reading agent. After all the state functions have

been defined, click “Create” button, then all the states functions are added to the “Monitors” drop-down list in the bottom part.

In the bottom part, the user is able to key in state values of the GUI state functions in the “Pre Monitor Result” text box. The GUI action name that is corresponding to the GUI state is entered in the “GUI action name” text box. If it is an input data field action, the user needs to key in the parameter in the “Parameter” text box. If this GUI action is the first field of an input data form, “Form Begin” checkbox should be checked. If the input filed is the last input field, “Form End” checkbox should be checked. The user also needs to give a form name.

After an action is entered, click button “Rule”. It indicates an action is finished.

After all the actions have been performed, click “Save All” button to save the results.

If the task of the application has already created before, the user could append new knowledge to an existing task.

6.1.2 Agents implementation

Five agents are implemented using Java language. Each agent extends JPanel and implements the java Runnable interface. The communications between agents use the java AWT events. Each plan defined in Chapter 4 is associated with a java AWT Event. Each plan is implemented as a java thread.

The screenshots of the GUI of each agent are shown in Figure 6-2.

The display panel of the monitoring agent is located at the left top corner. The display panel of the input data reading agent is located at the right top corner. The display panel of the playing agent is located at the left bottom corner. The display panel of the interface agent is located at the right bottom corner. The display panel of the reasoning agent is located at the center.

When the user wants to execute a task, he just keys in the task name and the application name (if applicable) through the interface of the GUI agent. The action to be executed is displayed at the playing agent’s display panel. The user simulates a GUI state by keying

in the state's value in the text box and by selecting radio button “E”, “S” or “C”, where “E” stands for unexpected exception, “S” stands for a GUI state message and “C” stands for the execution time of a GUI action that has elapsed.

Figure 6-2 Screenshot of agent interaction simulation application

6.1.3 Prototype testing

To test the prototype, we used Microsoft Outlook, Microsoft NotePad and our internal developed Staff Managing application as the applications to be tested. For each application we defined 10 tasks for processing normal, abnormal, and repeated situations. Here we use the internal developed Staff Managing application for demonstration.

We used two tasks according to the example GUI application shown in Figure 4-8 of Chapter 4. The two tasks are “adding user” and “deleting user”.

The actions sequence of the task “adding user” is shown as below:

```
New.click→Name.input("John")→Age.input("30")→Save.click→New.click→Name.input("Luc")
→Age.input("28")→Save.click→Exit.click
```

When browsing the user, if the age of the user is greater than eighty then delete the user. Otherwise, click next button to keep browsing. A state function "get_text" is used to get the value of user's age. The actions sequence of task "deleting user" is show as below:

```
Next.click( if getAge>=80 )→Delete.click→Next.click(if getAge<80)→Next.Click ( if getAge>=80 )→
Delete.click→Next.click ....
```

To test the prototype, we also defined an unexpected exception task, which has a state function "getErr" to get message in an "Error" dialog box. If the message is "No connection", then a "Retry" button is clicked. After that, a "resume" action is added to the task indicating the end of the unexpected task.

The three tasks were created by the capturing application. Rules & facts files, input data file, expected states file and unexpected state files were generated.

To test the "adding user" task, we first modified the input data file of the "adding user" task to include only one person, the GUI actions displayed in the panel of the playing agent were as expected. Then we modified the input data file of "adding user" task to include three persons, it worked as expected.

To test task "deleting user", we keyed in the state value of "get_text" by using value "81" and "79" and it worked as expected.

To test the unexpected exception, when performing task "deleting user" we selected radio box "U" and entered a value "No connection", String "Retry" was displayed on the playing agent's display panel. After that, the execution was resumed to task "deleting user".

The output (Jess rules) that was generated by the capturing application of task "deleting user" is listed as below:

The monitoring rule that was generated and associated with the task is as below:

```
(defrule DeletingUser _10
```

“Monitoring rule for task deleting user”

?id1<-(Start-monitor DeletingUser)

(State-action DeletingUser ?seq ?who ?method-name \$?para)

(State-action-num DeletingUser ?num)

=>

(Send-state DeletingUser ?seq ?who ?num ?id1 ?method-name \$?para)

)

The action rule and the monitoring rule that were generated for the action “Next.Click” is described as below:

(defrule DeletingUser _20

“Action rule for Next.Click”

?ida0<-(State-value DeletingUser GuiMonitor delay ?val0)

?idb0<-(Activation-value DeletingUser GuiMonitor delay DeletingUser 0 ?val0)

=>

(retract ?ida0)

(retract ?idb0)

(Play-action player Next.Click ?val0 not-data) ;

)

(defrule Next.Click _11

“Monitoring rule for Next.Click”

?id1<-(Start-monitor Next.Click) ; start the rule

(State-action Next.Click ?seq ?who ?method-name \$?para)

(State-action-num Next.Click ?num)

=>

(Send-state Next.Click ?seq ?who ?num ?id1 ?method-name \$?para)

)

The action rule and monitoring rule that were generated for action “delete” is described as below (“age>=80”):

(defrule Next.Click_21

“Action rule for Next.Click”

?ida0<-(State-value Next.Click GuiMonitor delay ?val0)

?idb0<-(Activation-value Next.Click GuiMonitor delay Next.Click1 ?val0)

?ida1<-(State-value Next.Click GuiMonitor get_text ?val1)

?idb1<-(Activation-value Next.Click GuiMonitor get_text Next.Click1 ?val1)

=>

(retract ?ida0)

(retract ?idb0)

(retract ?ida1)

(retract ?idb1)

(Play-action player Delete.Click ?val0 not-data)

)

(defrule Delete.Click_12

“Monitoring rule for Delete.Click”

?id1<-(Start-monitor Delete.Click) ; start

(State-action Delete.Click ?seq ?who ?method-name \$?para)

(State-action-num Delete.Click ?num)

=>

(Send-state Delete.Click ?seq ?who ?num ?id1 ?method-name \$?para)

)

The action rule and monitoring rule that were generated for action “Next” after action “Delete” is described as below:

(defrule Delete.Click_22

“Action rule for Delete.Click”

?ida0<-(State-value Delete.Click GuiMonitor delay ?val0)

?idb0<-(Activation-value Delete.Click GuiMonitor delay Delete.Click3 ?val0)

=>

(retract ?ida0)

(retract ?idb0)

(Play-action player Next.click ?val0 not-data)

)

(defrule Next.click_13

“Monitoring rule for Delete.Click”

?id1<-(Start-monitor Next.click)

(State-action Next.click ?seq ?who ?method-name \$?para)

(State-action-num Next.click ?num)

=>

(Send-state Next.click ?seq ?who ?num ?id1 ?method-name \$?para)

)

The action rule that was generated for action “next” is described as below (“age<80”):

```

(defrule Next.click_23

  “Action rule for Next.click”

  ?ida0<-(State-value Next.click GuiMonitor delay ?val0)

  ?idb0<-(Activation-value Next.click GuiMonitor delay Next.click4 ?val0)

  ?ida1<-(State-value Next.click GuiMonitor getAge ?val1)

  ?idb1<-(Activation-value Next.click GuiMonitor getAge Next.click4 ?val1)

  =>

  (retract ?ida0)

  (retract ?idb0)

  (retract ?ida1)

  (retract ?idb1)

  (Play-action player Next.Click ?val0 not-data )

)

```

Because the monitoring rule for action “next” when “age<80” is the same as rule *DeletingUser _20*. Therefore, this monitoring rule was discarded when it was added to the rule file. The above are all the rules generated for task “deleting user”.

The facts generated are listed as below:

```

(assert(State-action DeletingUser 0 GuiMonitor delay 0))

(assert(State-action-num DeletingUser 1))

(assert(State-action Next.Click 1 GuiMonitor delay 100))

(assert(State-action Next.Click 2 GuiMonitor getAge ))

(assert(State-action-num Next.Click 2))

(assert(State-action Delete.Click 3 GuiMonitor delay 100))

```

```
(assert(State-action-num Delete.Click 1))

(assert(State-action Next.click 4 GuiMonitor delay 100))

(assert(State-action Next.click 5 GuiMonitor getAge ))

(assert(State-action-num Next.click 2))
```

The expected state values that were extracted are described as below:

```
0 DeletingUser GuiMonitor delay DeletingUser 0 true

1 Next.Click GuiMonitor delay Next.Click1 true

2 Next.Click GuiMonitor get_text age Next.Click1 >=80

3 Delete.Click GuiMonitor delay Delete.Click3 true

4 Next.click GuiMonitor delay Next.click4 true

5 Next.click GuiMonitor get_text age Next.click4 <80
```

We could see from above, the GUI states (age ≥ 80 , age < 80) are extracted from the rules. The states values could be modified and so the rules could be reused.

6.2 Prototype implementation of GUI regression testing model

The GUI regression testing prototype is built based on JADE agent framework and jadex add-in is used to model each agent. Communications between agents uses **message events**. Events dispatched inside an agent uses **stimulus**.

To simplify the development, the KB managing agent is combined with the mediating agent and the remote capturing agent is combined with the playing agent. The recording agent could only record the mouse click events and input strings. It can not convert these events in component level. This does not affect the verification of agent interactions. These functionalities are beyond the scope of this research. To represent these events in component level, we manually created the GUI map for the GUI application and used this manually generated GUI map to translate the recorded low level mouse or keyboard events into component level events and for component level events to mouse and keyboard events.

In this prototype, one main JADE container and several JADE agent containers are started at the computers where the distributed GUI applications under test are running.

One recording agent and one GUI playing agent are initiated in the container where the distributed application under test is running. The mediating agent can be initiated in any container, and therefore it must be initiated before other agents initiate. The user interface agent and remote controlling agent run at the computer where the tester sits. Figure 6-3 shows an example of distribution of JADE containers. One main container and two agent containers are distributed in different computers. In Figure 6-3 the tester sits at the computer where the main container is running. There are four agents running in the main container. They are the mediating agent, the capturing agent, the playing agent and the remote controlling agent. The tester is able to operate the local GUI and two remote GUIs that are running at two different remote computers. There are two remote agent containers. In each remote container, two agents are running. They are the recording agent and the playing agent.

Figure 6-3 A scenario of agents' deployment

In this prototype, we define the following commands for the tester. These commands are invoked through the interface agent by the tester. They are listed as following:

- **new-test-case** *test_case_name application_name*. It is used to create a new test case which is represented by parameter *test_case_name* for a GUI application that is represented by parameter *application_name*
- **start-record** *test_case_name agent_name*. Begin recording the tester's actions using a capturing agent which is represented by parameter *agent_name*
- **stop-record** *test_case_name agent_name*. Tell the recording agent, which is represented by parameter *agent_name*, to stop recording a test case that is represented by parameter *test_case_name*.
- **start-capture** *capture_agent_name playing_agent_name*. Tell the remote capturing agent, which is represented by parameter *capture_agent_name*, to capture the screen and send it to the playing agent that is represented by parameter *playing_agent_name*.
- **stop-capture** *capture_agent_name playing_agent_name*. Tell the remote capturing agent, which is represented by parameter *capture_agent_name*, to stop capturing the screen.
- **start-play** *tsname appname*. Play a test case of an application.
- **stop-play** *tsname appname*. Stop playing a test case of an application
- **list-agents**. Display all the agents registered with the mediating agents.

Figure 6-4 shows the screenshot of the JADE remote monitoring agent's GUI. In the main container, a mediating agent, a remote controlling agent, a playing agent and an interface agent are running. In a remote computer, one recording agent and one playing agent are running. Each recording agent uses a separated container, because they are created by external Visual Basic application.

Figure 6-4 Screenshot of JADE remote monitoring agent

Figure 6-5 is the screenshot of remote screen displayed by the remote controlling agent, which is captured by the remote playing agent.

Figure 6-5 Screenshot of remote screen captured by the
remote controlling agent (left)

A test case can be created for both standalone GUI applications and distributed GUI applications. Test cases can also be played in the standalone and distributed environments. The implementations of the agents are listed as below:

6.2.1 Mediating agent

The mediator agent has the following belief:

b_1 =current_test_case. It is a type of string. It holds the current test case name being tested.

b_2 =application_name. It is a type of string. It holds the current application name being tested.

b_3 =registered_agents. It is a set of JADE AIDs. It holds the agents' ID registered with the mediator agent. Other agents in this model need to register themselves with the mediator agent.

The mediator agent has the following goals:

g_1 =df_register. This goal is to register the mediator agent to the JADE directory facilitator. Plan register_to_JADE_plan is started when the agent is active.

g_2 =process_commands. This goal is to process commands received from interface agent. Plan controlling_plan is started.

g_3 =create_test_case. This goal is to create a test case. When an internal event for creating a test case is generated from plan controlling_plan, plan create_test_case_plan is started.

g_4 =list_agents. This goal sends the agent IDs of all the registered agents to the interface agent. When an internal event for listing agents is generated from plan controlling_plan, plan list_agents_plan is started.

g_5 =agent_registry. This goal process the registry request from other agents. When an internal event for registry is generated from plan controlling_plan, plan registry_plan is started.

g_6 =start_recording. This goal lets the recording agent begin to capture user's actions. When an internal event for starting recording actions is generated from plan recording_control_plan, plan start_recording_plan is started.

g_7 =stop_recording. This goal lets the recording agent stop capturing user's actions. When an internal event for stopping recording actions is generated from plan recording_control_plan, plan stop_recording_plan is started.

g_8 =start_remote_capture. This goal lets the remote playing agent start capturing the remote screen. When an internal event for starting capturing the remote screen is generated from plan remote_capturing_control_plan, plan start_remote_capturing_plan is started.

g_9 =stop_remote_capture. This goal lets the remote playing agent stop capturing the remote screen. When an internal event for stop remote capturing the screen is generated from plan remote_capturing_control_plan, plan stop_remote_capturing_plan is started.

g_{10} =start_playing. This goal starts playing a test case. When an internal event for starting playing a test case is generated from plan playing_control_plan, plan start_playing_plan is started.

g_{11} =stop_playing. This goal stops playing a test case. When an internal event for stop playing a test case is generated from plan playing_control_plan, plan stop_playing_plan is started.

The mediator agent has the following plans:

p_1 =controlling_plan. This plan is started to accept commands from the interface agent and starts a new sub goal according to the command name.

p_2 =create_test_case_plan. This plan allocates memory for the new test case

p_3 =list_registered_agents_plan. This plan sends the agent IDs of the registered agents to the interface agent.

p_4 =register_to_JADE_plan. This plan registers the mediator agent with the JADE Directory Facilitator.

p_5 = recording_control_plan. This plan controls the starting and stopping of the recording agent. Sub goals for start_recording and stop_recording are started correspondingly.

p_6 =remote_capturing_control_plan. This plan switches playing and screen capturing mode of a playing agent.

p_7 =playing_control_plan. This plan controls the starting and stopping of the playing agent. Sub goal for start_playing and stop_playing goals are started correspondingly.

p_8 =start_recording_plan. This plan sends a message to the capturing agent and let the capturing agent begin to capture the tester's actions.

p_9 =stop_recording_plan. This plan sends a message to the capturing agent and let the capturing agent stop capturing the tester's actions

p_{10} =start_remote_capturing_plan. This plan sends a message to the remote playing agent and let the remote playing agent begin to capture the remote screen.

p_{11} =stop_remote_capturing_plan. This plan sends a message to the remote playing agent and lets the remote playing agent stop capturing the remote screen.

p_{12} =start_playing_plan. This plan sends a message to the playing agent and let the playing agent start playing a test case.

p_{13} =stop_playing_plan. This plan sends a message to the playing agent and let the playing agent stop playing a test case.

p_{14} =registry_plan. This plan saves the ID of the agent which is requesting for registry to its belief base.

6.2.2 Interface agent

The interface agent has the following beliefs:

b_1 =last_received_msg. it is a string type. It holds the message just received.

The interface agent has the following goals:

g_1 =display_message. It displays the received message to its GUI. When an internal event is generated from plan receive_message_plan, plan display_message_plan is started.

g_2 =receive_message. It receives message to be displayed. When a message is received from the mediator agent, plan receive_message_plan is started.

The interface agent has the following plans:

p_1 =display_message_plan. This plan displays the message to the message panel of the agent.

p_2 = receive_message_plan. This plan receives the message from the mediator agent. Sub goal display_message is started.

6.2.3 Recording agent

The recording agent has the following beliefs:

b_1 =test_case_name. It is a string type. It holds the test case name being captured.

b_2 =mediator_agent_id. It is a JADE AID type. It holds the agent ID of the mediator agent.

The recording agent has the following goals:

g_1 =register_mediator. This goal registers it with the mediator agent. This goal is started when the agent is active.

g_2 =df_serach. This goal searches the mediator agent. When an internal event is generated from plan register_with_mediator_plan, plan df_search_plan is started.

g_3 =receive_commands. This goal receives commands from the mediator agent. When a command is received from the mediator agent, plan receive_commands_plan is started.

g_4 =send_actions. This plan sends captured actions to the mediator agent. Plan send_actions_plan is started when the user performs an action.

The recording agent has the following plans:

p_1 =df_search_plan. This plan searches the mediator agent from the Directory Facilitator of JADE.

p_2 =register_with_mediator_plan. This plan registers the recording agent with the mediator agent. Sub goal df_search is started to search the mediator agent. The belief base is also updated if the mediator agent is found.

p_3 =receive_commands_plan. This plan receives commands from the mediator agent. If it is a start_recording command then the agent begins to record user's actions. If it is a stop_recording command, the agent stops the recording.

p_4 =send_actions_plan. This plan sends the recorded GUI action to the mediator agent.

6.2.4 Playing agent

The playing agent has the following beliefs:

b_1 =capture_play_switch. It is a string type. It holds the switch state. If it is "ON" then the agent just plays GUI actions. If it is "OFF", the agent will capture the screen after have played a GUI action.

b_2 =remote_controlling_agent_id. It is a JADE AID type. It holds the agent ID of the remote controlling agent.

The playing agent has the following goals:

g_1 = playing_action. This goal plays a GUI action. When a GUI action is received from the mediator agent, plan playing_action_plan is started.

g_2 =capturing_screen. This goal captures the screen. When a message is received from the remote controlling agent, plan capturing_screen_plan is started.

The playing agent has the following plans:

p_1 =playing_action_plan. This plan converts the GUI action into mouse and keyboard events and sends them to the underlying system.

p_2 =capturing_screen_plan. This plan captures the screen and sends it to the remote controlling agent.

6.2.5 Remote controlling agent

The remote controlling agent has the following beliefs:

b_1 =playing_agent_id. It is a type of JADE AID. It holds the agent ID of the remote playing agent.

b_2 =sx. It is a double. It holds the scale rate of the width of the captured screen.

b_3 =sy. It is a double. It holds the scale rate of the height of the captured screen.

b_2 and b_3 are used to convert the captured screen to a specific size.

The remote controlling agent has the following goals:

g_1 =show_screen. This goal displays the captured screen on the GUI of the agent. It is started when an internal message is received from plan receive_message_plan.

g_2 =receive_screen. This goal receives a captured screen from the remote playing agent. When a message from the remote playing agent is received, plan receive_screen_plan is started.

g_3 =send_event. This goal sends the mouse and keyboard events to the remote playing agent. Plan send_event_plan is started when the user clicks the mouse or presses a key.

The remote controlling agent has the following plans:

p_1 =show_screen_plan. This plan displays the captured screen from the remote playing agent.

g_2 =receive_screen_plan. It first transforms the screen to the size of the GUI of the remote controlling agent. It then starts sub-goal show_screen to display the captured screen.

g_3 =send_event_plan. This plan sends the mouse and keyboard events to the remote playing agent.

6.2.6 Prototype testing

To test the model, we select Yahoo Messenger, Microsoft NotePad and our internal developed Staff Managing application as the applications to be tested. We created 30 test cases manually and use the prototype to create, execute and repair test cases. These test cases were represented in AKG and SGLs correctly. These test cases were able to be reconstructed and executed. AKG and SGLs were updated when a GUI element is modified and test cases could be reconstructed correctly. For example, to test the modification, we modified the form in the Staff Managing application by removing the “Exit” button, instead “x” button is used to exit. After test case “add new staff” was repaired. Test case “add existing staff” was also repaired automatically.

The testing was conducted on a 1.4 G Pentium M laptop with 512M of RAM. The average length of the 30 test cases was 7 actions. The total time of constructing the AKG and SGL was 300ms. The average time of finding a test case to which an action belongs was 2.86 ms. The average time of deleting an action from the AKG and SGL was less than 1 ms. The average time of adding a new edge to the AKG and SGL was less than 1 ms.

6.3 Summary

In this chapter we introduced the two prototypes developed to verify the two models which we proposed in Chapter 4 and Chapter 5. The two developed prototypes shows that the two proposed models could work as expected.

Chapter 7 Conclusion and Recommendations

7.1 Conclusion

This thesis first discussed a general intelligent agent based automatic operating models to be applied to automate GUI application. In this thesis, we identified two types of GUI application: product applications and applications under test. Since there is a big difference between product applications and applications under test, two separated models are proposed. In the GUI automation model, we presented rules based knowledge representation for GUI actions and runtime context. In the GUI regression testing model, we presented an actionable knowledge model to represent test cases. The GUI automation model enables different actions to be taken according to the GUI execution context and it has the mechanism to handle unexpected exceptions. The GUI regression testing model makes it possible to repair one test case and make all the other test cases to get repaired auto/matically. It applies multi-agent technologies to facilitate the test case creation, execution and modification. It has the ability to test distributed GUI applications.

The nature of GUI automation and GUI regression testing are different. GUI automation is oriented to gather knowledge through the use of GUI applications. GUI regression testing is oriented to find bugs in GUI applications in the development phase. Due to the difference between GUI automation and GUI regression testing, we have dealt with them separately in this thesis. We have presented two new separate models for GUI automation and regression testing.

In this thesis, a rule based representation for the GUI states (contexts) and its corresponding user's actions are used. Two types of rules are generated for each user's action and its corresponding execution context. One of them is the monitoring rule and the other one is the action rule. The monitoring rule is used to send all the state functions to the monitoring agent or input data reading agent (if any). The action rule is used to retrieve a GUI action when the state of the running GUI application matches a rule's "if" part.

In the GUI automation model, the capturing application is used to teach the automation system. All the user's actions are captured. The user could define the GUI states which cause this GUI action. If there are no explicit GUI states are defined by the user, the completion a GUI action is used as the state. The capturing application translates this information into rules. The states are stored in text file. The input data is also extracted from the rules and stored in a text file. The user could modify these text files to apply different states and input data to reuse the generated rules without modifying the rules.

In the GUI automation model, five agents are defined. At automation phase, these agents work together to drive GUI application to perform a task automatically. Whenever a known unexpected happens, these agents could deal with this unexpected exception first and then resume the interrupted task.

In the GUI regression testing model, seven agents are defined in the model.

The tester could issue commands through interface agent to control all the agents distributed in different computers to perform testing related activities. The tester is also able to operate the GUI of a remote GUI application through the remote controlling agent. All the tester's actions (a test case) are captured in component level and stored in the form of AKG and SGL. At a later time the tester could play the recorded test case in a distributed environment.

The tester is able to repair test cases using an incremental way when there are changes of the GUIs and these changes invalidate some of the test cases. The tester can tell the system a name of a GUI element. The system can execute the GUI application to the area of the modified GUI automatically. Then the tester can manually re-record the invalid part of the test case. The system is able to find all the test cases affected by the modification or by the repair itself. The tester could decide if to do further repair.

All the test cases of a GUI application under test are stored in the same AKG. Each test case also has a SGL, which list all the goal-vertices in the test case. Test case reconstruction is based on the AKG and the test case's SGL.

7.2 Recommendations for further research

The representations, which are used in both GUI automation and GUI regression testing model, assume each of the user's action could be represented in the component level. That means each event must be assigned to a GUI component. If the capturing agent could capture the user's event, represent this event as a GUI component's event and the playing agent is also able to interpret the same event correctly, our agent model is able to work correctly. But for mouse motion events, the current tools use analog recording to record the precise x- and y-coordinates traveled by the mouse pointer across the screen. So research is needed to investigate how to represents the mouse motion in component level.

The proposed automation model for product GUI application is not able to be applied to the distributed environment. Further research could extend it to the distributed environment by referencing the regression testing model.

The prototype of the GUI regression testing model did not implement the capturing agent, playing agent and remote screen capturing agent as mobile agents. Further implementation could implement these agents as mobile agent and add the ability to install the system dependent components that are used to capture and replay user events after these mobile agents has moved to a new location.

This research is not isolated. It is useful to explore how other areas of software engineering could benefit from this research, for example, Software Configuration Management (SCM). SCM tools track which versions of modules to be included in a particular system build and which tests to be run. SCM tools also track the results of the corresponding tests. To Link this research to SCM is useful. If protocols are established between the two types of systems, test cases and defects reports generated in this research are possibly imported and exported to a SCM tool for easy management. There could be two ways to start, the first way is to treat the current SCM tools as "as is". New goals and plans could be added to our mediator agent as proposed in Section 5.5. A communication bridge may also need to be built between the two systems. Another possible way is to build a fully integrated system from scratch. This could become complicated. For example, if a defect is identified in the testing phase, the requirement may need to be changed to address this problem. Change in a requirement will affect

Chapter 7 Conclusions and Future Work

design and coding, and a new test case needs to be created and to be export to this agent system.

AUTHOR'S PUBLICATIONS

BIBLIOGRAPHY

- [1] A. M. Memon and M. L. Soffa, "Regression testing of GUIs". Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering, Vol. 28, No. 5, pp. 118-127, Sept 2003.
- [2] Wen, R. B, "URL-driven automated testing". Second Asia-Pacific Conference on Quality Software, pp. 268–272, Dec 2001.
- [3] A. M. Memon, M. L. Soffa, and M. E. Pollack, "Coverage criteria for GUI testing". Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, Vol. 26, No. 5, pp. 256-267, Sept 2001.
- [4] D. J. Kasik and H. G. George, "Generating test cases for GUI responsibilities using complete interaction sequences". Proceedings of the International Symposium on Software Reliability Engineering, pp. 110-121, Oct 2000.
- [5] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Automated test oracles for GUIs", Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering, Vol. 25, No. 6, pp. 164-173 ,Nov 2000.
- [6] J. A. Whittaker, "What is software testing? And why is it so hard?". IEEE Software, Vol. 17, No. 1 , pp. 70 -79, Jan/Feb 2000.
- [7] Wahi, Y. J. Li, and J. Nancy, "An overview of regression testing". ACM SIGSOFT Software Engineering Notes, Vol. 24, No. 1, pp. 69-73, Jan 1999.
- [8] WinRunner, test automation for the enterprise.
<http://www.mercuryinteractive.com/products/winrunner/> (Jan-2004)
- [9] Rational Robot.

<http://www-306.ibm.com/software/awdtools/tester/robot/> (Jan-2004)

[10] H. R. bhambhabi, "An expert system for suggesting design patterns". thesis, University of Texas at Arlington, 2002

[11] H. S. Nwana, "Software Agent: an Overview, Knowledge engineering Review". Vol. 11, No. 3, pp. 1-40, Sept 1996.

[12] What is a Software Agent?

<http://agtivity.com/agdef.htm>

[13] D. W. Croft, "Intelligent Software Agents: Definitions and Applications". <http://www.alumni.caltech.edu/~croft/research/agent/definition/>

[14] T. I. Zhang, E. Kendall, and H. Jiang, "A software engineering process of BDI agent-based systems". IEEE/WIC International Conference on Intelligent Agent Technology, pp. 392–399, Oct 2003.

[15] J. Thangarajah, L. Padgham, and J. Harland, "Representation and reasoning for goals in BDI agents". Proceedings of the twenty-fifth Australasian conference on Computer science, Vol. 24, No. 1, pp. 259-265, Jan 2002.

[16] Y. Labrou, T. and Finin, and Y. Peng, "The current landscape of Agent Communication Languages". IEEE intelligent systems, Vol. 14, pp.45-52. Mar/Apr 1999.

[17] M. R. Genesereth and S. P. Katchpel, "Software agents". CACM, vol. 37, No. 7, pp. 48–53, 1994.

[18] "FIPA Agent Communication specifications"

<http://www.fipa.org/repository/aclspecs.php3>

[19] A. Rao and M. George, "Modeling rational agents within a BDI architecture". Proceedings of the Second International Conference on Knowledge Representation and

Reasoning, pp. 473–484. Morgan Kaufmann, Feb 1991.

[20] Ostrand, A. Anodide, H. Foster, and T. Goradia, “A visual test development environment for GUI systems”. Proceedings of the A CM SIGSOFT International Symposium on Software Testing and Analysis, pp. 82-92, New York, ACM Press, Mar1998.

[21] B. Riemsdijk, W. Hoek, and J. J. Ch. Meyer, “Agent decision making: Agent programming in dribble: from beliefs to goals using plans”. Proceedings of the second international joint conference on Autonomous agents and multiagent systems, pp. 393-400, July 2003.

[22] P. R. Cohen and H. J. Levesque, “Intention is choice with commitment”. Artificial Intelligence, Vol. 42, pp.213–261, Mar 1990.

[23] M.J. Wooldridge and N.R. Jennings, “Agent theories, architectures, and languages: a survey”. Proceedings of the ECAI-94 Workshop on Agent Theories, Architectures, and Languages, vol. 890, pp.1–25. Springer-Verlag, Aug 1994.

[24] M. George., B. Pell, M.E. Pollack, M. Tambe, and M. Wooldridge, “The Belief-Desire-Intention model of agency”. Intelligent Agents V, vol. 1365, pp. 1-10, Mar 1999.

[25] M. P. Singh, A. S Rao, and M. P. George, “Formal methods in DAI: Logic-based representation and reasoning”. Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence, Chapter 8, pp. 331–376. MIT Press, 1999.

[26] M. E. Bratman. “Intentions, Plans, and Practical Reason”. Harvard University Press, Cambridge, MA, pp.140, 1987.

[27] C. Heinze, B. Smith, and M. Cross, “Thinking quickly: Agents for modeling air warfare”. Proceedings of 4th Australian Joint Conference on Artificial Intelligence. AI’98, pp. 47-58, July 1998.

[28] F. Dignum, M.D. Morley, E. A. Sonenberg, and L. Cavedon, “Towards socially sophisticated BDI agents”. Proceedings of Fourth International Conference on

MultiAgent Systems, pp. 111-118, July 2000.

[29] P. Balbiani and A. Elfallah-Seghrouchni, "A modal logic for reasoning about beliefs, desires and intentions of complex agents in a distributed environment ". Proceedings of International Conference on Multi Agent Systems, pp. 387 –388, July 1998.

[30] M. Winicoff, L. Padgham, and J. Harland, "Simplifying the development of intelligent agents, Advances in Artificial Intelligence". 14th Australian Join Conference on artificial intelligence, pp. 555-568, Dec 2001

[31] M. Papasimeon and C. Heinze, "Extending the UML for designing JACK agents". Proceedings of the Australian Software Engineering Conference (ASWEC 01), pp. 89-97, Aug 2001.

[32] J. Tavares and Y. Demazeau, "Vowels co-ordination model". pp.1129-1136, AAMAS, July 2002.

[33] M. Tambe, D. Pynadath, and N. Chauvat, "Building dynamic agent organizations in cyberspace". IEEE Internet Computing, Vol. 4, No. 2, pp.65-73, Mar/Apr 2000.

[34] G. Tidhar, "Team-oriented programming: Social structures". Technical Report 47, Australian A.I. Institute, 1993.

[35] JADE Agent Platform

<http://sharon.cselt.it/projects/jade/>

[36] B. A. Myers, J. D. Hollan, and I. F. Cruz, "Strategic directions in human-computer interaction". A CM Computing Surveys, Vol. 28, No.4, pp. 794-809, Dec 1996.

[37] T. Daboczi, I. Kollar, G. Simon, and T. Megyeri, "How to test graphical user interfaces". IEEE Instrumentation & Measurement Magazine, Vol. 6 , No. 3 , pp. 27-33, Sept 2003

[38] A. M. Memon, I. Banerjee, and A. Nagarajan, "What test Oracle should I use for effective GUI testing?". Proceedings on Automated Software Engineering, pp. 164 –

173, Oct 2003.

[39] Z. Guessoum and J. P. Briot, "From Active Object to Autonomous Agents". IEEE Concurrency, pp. 68- 78, July/Sept 1999.

[40] C. L. Hsu; S. F. Chou; J. C. Tsay; and F. J. Wang, "JTEAM: A framework for effective teamwork of BDI-based agents". Proceedings of The Ninth IEEE Workshop on Distributed Computing Systems, pp. 190 –196, May 2003.

[41] B.A. Myers, "User Interface Software Tools". ACM Transactions on Computer-Human Interaction, vol. 2, no. 1, pp. 64-103, Mar 1995.

[42] A. M. Memon, "A Comprehensive Framework for Testing Graphical User Interfaces". Ph.D. thesis, Department of Computer Science, University of Pittsburgh, 2001.

[43] B. A. Myers, "Why are Human-Computer Interfaces Difficult to Design and Implementation". Technical Report CS-93-183, School of Computer Science, Carnegie Mellon Univ., July 1993.

[44] W. I. Wittel Jr. and T. G. Lewis, "Integrating the MVC Paradigm into an Object-Oriented Framework to Accelerate GUI Application Development". Technical Report 91-60-06, Dept. of Computer Science, Oregon State Univ., Dec 1991.

[45] P. T. Graunke, and S. Krishnamurthi, "Advanced control flows for flexible graphical user interfaces". Proceedings of the 24th International Conference on Software Engineering, pp. 277-287, May 2002

[46] J. Chen, and S. Subramaniam, "A GUI environment to manipulate FSMs for testing GUI-based applications in java". Proceedings of the 34th Annual Hawaii International Conference on System Sciences, pp. 3915 –3924, Jan 2001

[47] A. M. Memon, "GUI testing: Pitfalls and process". IEEE Computer, Vol. 35, No. 8, pp.90–91, Aug 2002.

[48] A. M. Memon, M. E. Pollack, and M. L. Sofia, "Using a goal-driven approach to

generate test cases for GUIs”. Proceedings of the 1st International Conference on Software Engineering, pp. 257-266. ACM Press, May 1999.

[49] A. M. Memon, M. E. Pollack, and M. L. Soffa, “Hierarchical GUI Test Case Generation Using Automated Planning”. IEEE Transactions on Software Engineering, vol. 27, no. 2, pp.144-155, Feb 2001.

[50] K. Shehady and D. P. Siewiorek, “A method to automate user interface testing using variable finite state machines”. Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing, pp. 80-88, Washington-Brussels - Tokyo, IEEE Press, June 1997.

[51] E. J. Weyuker, “The applicability of program schema results to programs”. International Journal of Computer and Information Sciences, Vol. 8, No. 5, pp. 387-403, Oct 1979.

[52] L. Cholvy, and C. Garion, “Coordination in MAS: Distribution of goals addressed to a group of agents”. Proceedings of the second international joint conference on Autonomous agents and multiagent systems, pp. 765-772, July 2003.

[53] J. H. Hicinbothom and W. W. Zachary, “A tool for automatically generating transcripts of human-computer interaction”. Proceedings of the Human Factors and Ergonomics Society 37th Annual Meeting, vol. 2, pp. 1042, SPECIAL SESSIONS: demonstrations, 1993.

[54] A. Kaster, “User Interface Design and Evaluation—Application of the Rapid Prototyping Tool”. Proceeding on Human-Computer Interaction, vol. 1, pp. 635-639, 1991.

[55] J. Nielsen, “Iterative User-Interface Design”. Computer, vol. 26, no. 11, pp. 32-41, Nov 1993.

[56] M. M. Kaddah, “Interactive Scenarios for the Development of a User Interface Prototype”. Proceeding on Human-Computer Interaction, vol. 2, pp. 128-133, 1993.

[57] D. Rosenberg, “User Interface Prototyping Paradigms”. Conference on Human

Factors in Computing Systems, pp. 231, 1993.

[58] M. G. El-Said, G. Fischer, S. A. Gamalel-Din, and M. Zaki, “A Tool for Automating the Design of Visual Interfaces”. Computers & Graphics, vol. 21, no. 1, pp. 79-87, 1997.

[59] B. A. Myers, D. R. Olsen, Jr., and J. G. Bonar, “User interface tools”. Proceedings of A UM INTERCH Conference on Human Factors in Computing Systems, Tutorials, pp. 239, Apr 1993.

[60] L. White, H. Almezen, and N. Alzeidi, “User-based testing of GUI sequences and their interactions”. Proceedings of 12th International Symposium on Software Reliability Engineering, pp. 54-63, Nov. 2001.

[61] L. White, “Regression Testing of GUI Event Interactions”. Proceeding on Software Maintenance, pp. 350-358, Nov 1996.

[62] D. Binkley, “Semantics guided regression test cost reduction”. IEEE Transactions on Software Engineering, Vol. 23, No. 8, pp. 498–516, Aug. 1997.

[63] D. S. Rosenblum and E. J. Weyuker, “Using coverage information to predict the cost-ectiveness of regression testing strategies”. IEEE Transactions on Software Engineering, Vol. 23, No. 3 pp.146–156, Mar 1997.

[64] G. Rothermel and M. J. Harrold, “A safe, efficient regression test selection technique”. ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 2, pp.173–210, Apr 1997.

[65] G. Rothermel and M. J. Harrold, “Empirical studies of a safe regression test selection technique”. IEEE Transactions on Software Engineering, Vol. 24, No. 6, pp. 401–419, Jun 1998.

[66] R. S. Pressman, “A Practitioner’s Approach”. Software Engineering, McGraw-Hill, 3rd edition, 1994.

[67] B. Beizer, “Software Testing Techniques”. Van Nostrand Reinhold, New York,

2nd edition, 1990.

[68] Fewster, Mark, “Software Test Automation”. Addison Wesley, New York, 1999.

[69] J. D. Ruvini, and C. Fagot, BHYS: “A new approach to learn user’s habits”. Tenth IEEE International Conference on Tools with Artificial Intelligence, pages 200 –207, Nov. 1998.

[70] K. C. Bourne, “Client/Server Testing Tools”

<http://www.dbmsmag.com/9606d16.html>