Analysing Reverse Engineering Techniques for Interactive Systems

A thesis

submitted in partial fulfillment

of the requirements for the degree

of

Master of Science in Computer Science

at

The University of Waikato

by

Feifei (Amy) Lin

supervised by

Judy Bowen・Steve Reeves

The University of Waikato

Department of Computer Science

Hamilton, New Zealand

2012

Abstract

Reverse engineering is the process of discovering a model of a software system by analyzing its structure and functions. Reverse engineering techniques applied to interactive software applications (e.g. applications with *user interfaces* (UIs)) are very important and significant, as they can help engineers to detect defects in the software and then improve or complete them. There are several approaches, and many different tools, which are able to reverse-engineer software applications into formal models. These can be classified into two main types: dynamic tools and static tools. Dynamic tools interact with the application to find out the run-time behaviours of the software, simulating the actions of a user to explore the system's state space, whereas static tools focus on static structure and architecture by analysing the code and documents. Reverse engineering techniques are not common for interactive software systems, but nowadays more and more organizations recognize the importance of interactive systems, as the trend in software used in computers is for applications with graphical user interfaces. This has in turn led to a developing interest in reverse engineering tools for such systems.

Many reverse engineering tools generate very big models which make analysis slow and resource intensive. The reason for this is the large amount of information that is generated by the existing reverse engineering techniques. Slicing is one possible technique which helps with reducing un-necessary information for building models of software systems. This project focuses on static analysis and slicing, and considers how they can aid reverse engineering techniques for interactive systems, particularly with respect to the generation of a particular set of models, *Presentation Models* (PModels) and *Presentation Interaction Models* (PIMs).

Acknowledgments

I would like to express my great appreciations to my supervisors, Dr. Judy Bowen and Professor Steve Reeeves. I am grateful for their support, encouragement, expert guidance and commitment to this thesis. Their comments and proofreading help me to finish this thesis. I would also like to thank the librarian, Ms. Cheryl Ward, for her patience with checking through the bibliography and helping me to improve it. Also, thanks to Peilun Hu for her occasional massage to relieve my stiff shoulders! Finally, huge gratitude to my wonderful parents who gave me enormous love and freedom, my siblings who encouraged me, and my lovely 3-year-old nephew who always make me happy. Without these people, this thesis would never have happened.

Contents List

List of Tables

List of Segments

List of Figures

Chapter 1

Introduction

1.1 Background

Interactive systems have been developed for years and are widespread in the information age. Lots of applications with *user interfaces* (UIs) are used on computers, for example, *Graphical user interface* (GUI) applications. GUIs play a very important role in today's software by making the operation of applications more visually oriented with basic objects such as menus, buttons etc. It is easier for people to understand those applications than applications with command line interfaces, and they can more easily access the system functionality (which refers to the internal behaviours of the system). Enterprise competitiveness is very much dependent on the quality of the GUI because good-quality of the GUIs ensures usability of the systems. However, interactive systems often contain design defects that may cause users problems [9] and which are hard to find, and it is difficult to maintain the quality of complex and large user interfaces. A large proportion of failures in interactive systems are caused by human error, as described by Leveson [1]. Human error in computer system use is often due to errors in the user interface design and not the sole result of errors of the users of the systems.

As part of software development, we should ensure we correctly understand what the client's requirements of the software are. This means that when we design it, the requirements will always be met (assuming we design it correctly), and it can also be guaranteed that when we turn the design into implementations the requirements will be maintained (assuming we follow relevant formal processes). There is no guarantee that knowing the requirements ensures programmers will program in strict accordance with them. We need proof of this before we implement them.

An approach we rely on to achieve this is to use formal methods, which can be defined as follows:

> "*we follow a process which uses some formal language to specify the behaviour of the intended system, techniques such as theorem proving or model-checking to ensure the specification is valid (i.e. meets the requirements and has been shown, perhaps by proof or other means of inspection, to have the properties the client requires of it) and a refinement process to transform the specification into an*

implementation" [4].

Another reason to use formal methods is that they help with describing the functionality of systems (application logic or underlying system behaviour) and this is useful because it acts as a bridge between the implementation and the artifact design, giving a clear idea about functionalities of systems for programming which streamlines the development process.

We can describe how we will consider the components of a software system by the illustration in Figure 1.

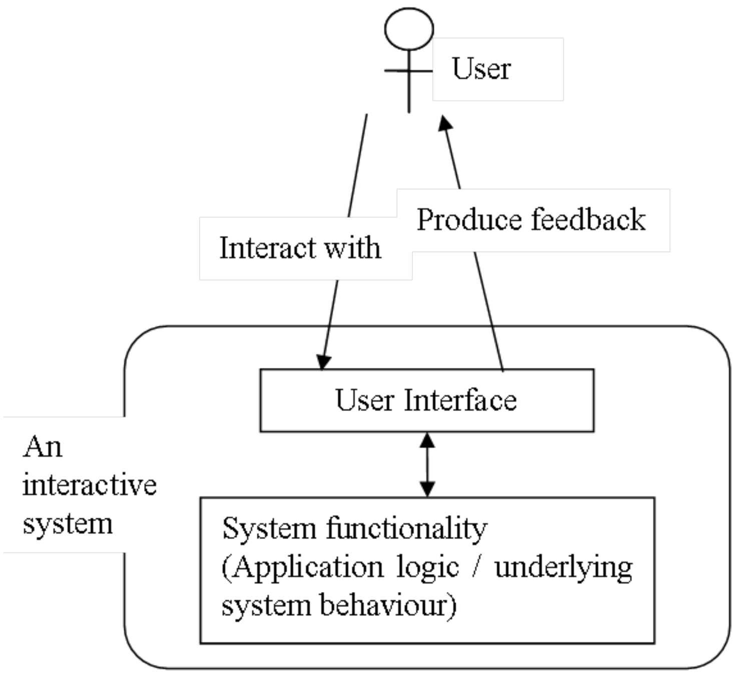

User
Interact with
Produce feedback
User Interface
An interactive system
System functionality (Application logic / underlying system behaviour)

Figure 1: The components of an interactive system and the interaction between a user and an interactive system.

An interactive system is constructed from a user interface and the system functionality. The system functionality describes all of the internal behaviours of a system. The user interface allows a user to access the system functionality. Many existing well-known reverse engineering techniques and tools focus on the underlying system behaviour (system functionality), such as the model-to-implementation mapping tool used in [10] to check if an implementation of a software system conforms to the specification of that system. Recently, there have also been some tools developed to reverse engineer graphical user interfaces, and collect information on a GUI's structure, like GUI Ripper [7]. What we want to do is reverse engineer the system to get both the structure of the GUI and its corresponding internal behaviour. We illustrate this in Figure 2.

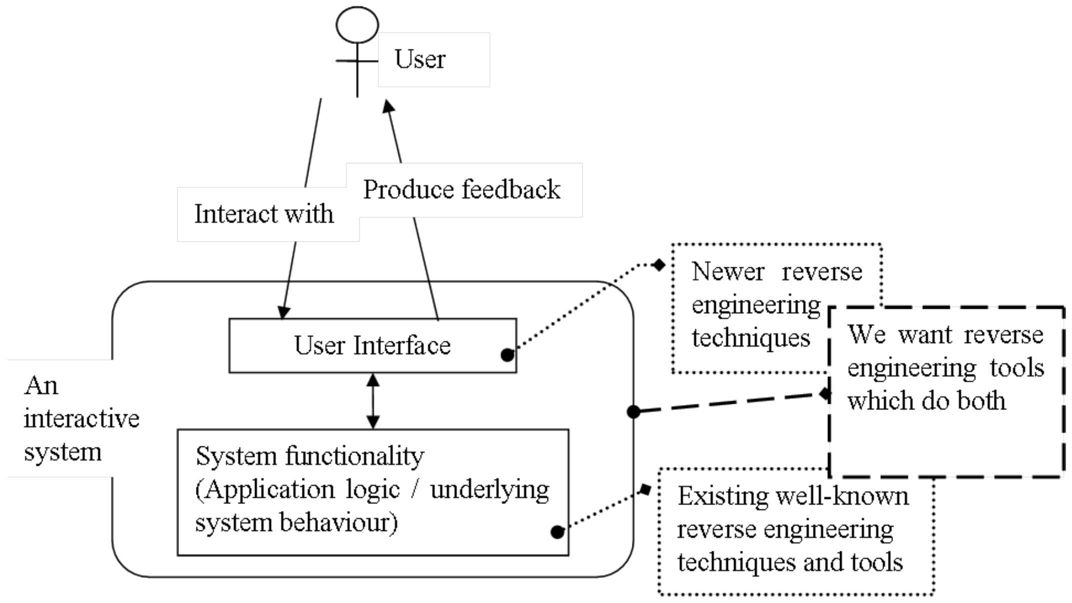

User
Interact with
Produce feedback
Newer reverse engineering techniques
We want reverse engineering tools which do both
User Interface
An interactive system
System functionality (Application logic / underlying system behaviour)
Existing well-known reverse engineering techniques and tools

Figure 2: The different emphasis of different reverse engineering techniques for an interactive system.

Meeting the client's requirements is not enough for developing software. We must also consider the user, to ensure they are able to satisfactorily use the software we have built. Apart from the correctness of the underlying application, good-quality software must meet the satisfaction of the user.

An approach named *User-Centered Design* (UCD) tackles these concerns and involves the user at the centre of the design process.

> "*User-centered design can be characterized as a multi-stage problem solving process that not only requires designers to analyze and foresee how users are likely to use a product, but also to test the validity of their assumptions with regards to user behaviour in real world tests with actual users*" [5].

User requirements are considered right from the beginning and throughout the whole process. These requirements are noted and refined through some methods like

> "*ethnographic studies, brain-storming sessions using white-boards and post-it notes, paper and pencil sketches etc. and are intended to convey information quickly and easily to non-technical people, i.e. real users rather than software developers*" [4].

This approach tries to ensure that the system we build is usable by the end-users. UCD is strongly related to *Human-Computer Interaction* (HCI).

HCI is the study of the interaction between the user and the computer. It focuses on the design, evaluation, and implementation of interactive computer systems which are to be used by humans. This approach does not specify any order to the activities surrounding this evaluation. It strongly iterates user evaluation throughout the whole process, adapting to the needs of the users while issues are discovered or changed. Therefore, a good design of user interface is created, meeting the expectations of the intended users.

In order to ensure correct execution of the overall software, the correctness of the user interface is essential. The UI's correctness includes its usability, which covers the effectiveness, efficiency, and satisfaction that is the basic and main characteristic. This characteristic allows the user to interact with the system to achieve their goals. In order to have a good user interface it must both be properly designed and properly implemented.

Reverse engineering techniques are a popular and meaningful way to detect defects in software applications and then improve or complete them. Reverse engineering is the process of analyzing the structure, function, and operation of a device / object / system to discover its technological principles. In the field of software development, there are various uses of reverse engineering but no consistency with the terminology. Chikofsky and Cross researched the uses and defined a taxonomy in their paper [3], which states,

"R*everse engineering is the process of analyzing a subject system to* *identify the system's components and their interrelationships, and create representations of the system in another form or at a higher level of abstraction*".

We can understand reverse engineering as the initial examination of the system by analyzing an existing software system and then inferring its design. Re-engineering is the subsequent modification of the system generally by adding new functionality or correcting errors in the design. Software re-engineering is mostly used for legacy systems to better fit the needs, for example, of the different environments required. Some examples of reverse engineering uses include those given in Samir [8], who migrates desktop applications like Java-Swing applications to the web as Ajax applications. Also mentioned in [8], is the famous mobile brand Nokia who use WebCream, a commercial tool for a Java to HTML conversion that constructs HTML front ends for Java applications to web-enable their networking software. These are both examples of commercial re-engineering methods based on reverse engineering principles.

Model-based design helps designers to specify and analyze systems through identifying high-level models. Prior to implementation, the models are then verified and used as the basis for the development of the implemented software. Several tools are able to reverse-engineer software applications into formal models. In practice, these tools can be classified into two main types: dynamic tools and static tools. Dynamic tools interact with the application to find out the run-time behaviours of the software, simulating the actions of a user to explore the system's state space. For example, the GUIRipper tool developed by Memon et al. [7] opens all of the *software under test* (SUT)'s windows, and extracts all widgets, properties, and values using a dynamic process; whereas static tools focus on static structure and architecture by analyzing the code and documents, e.g. Corbett [6] extracted a finite state model from Java source code using the Bandera tool. The basic difference between them is that source code of the re-engineered application is required for the static tools but not necessarily for dynamic tools. This has implications in terms of the information that can be gathered.

In this project, I examine reverse engineering techniques which analyse the structure and behaviour of interactive software applications, with the aim of identifying how such tools could be used to build a particular set of models, *Presentation Models* (PModels) and *Presentation Interaction Models* (PIMs). These models were developed by Bowen and Reeves [4] with the intention of creating formal descriptions of interactive systems. The *presentation model* is used to formally capture the meaning of the user interface of a system or an informal design artifact such as a scenario, storyboard or prototype, the *presentation interaction model* provides a view of the dynamic changes of the UI. The combination of PModels with μCharts is called PIM and we describe these using the μCharts language [37]. I will describe these models in detail in Section 4.2.

1.2 Defining the Problem and Outline of Possible Solution

This project examines reverse engineering techniques and how they can be used for interactive software applications. It shows how both types (static and dynamic analysis) when used in this context have shortcomings.

Static analysis involves considering the software's code. There may be thousands of lines of code for an application, so attempting this manually (i.e. by reading and understanding the code) is not a practical approach. In this case, some tools can help. These may be based on reverse engineering principles such as *Parsing* and *Slicing*. We can use a parser to generate the *Abstract Syntax Tree* (AST) of the software and then use slicing tools to reduce the entire AST and then get the parts of the AST only relating to the GUI. This reduces the amount of code which needs to be considered. However, it sometimes causes some problems, such as the ability to understand some of the information or missing hidden information, and so on. We will give more details of this in Chapter 6.

The AST models an entire representation of the abstract syntactic structure of source code written in a programming language (e.g. Java) in a tree form. A tree representation of a Java source code is revealed as shown in Figure 3.

Figure 3 shows that the AST is good for the static reverse engineering as it breaks the source code into useful components that means we can get the individual information of each component. However, it means we need a way to traverse the tree to find the relevant parts we care about.

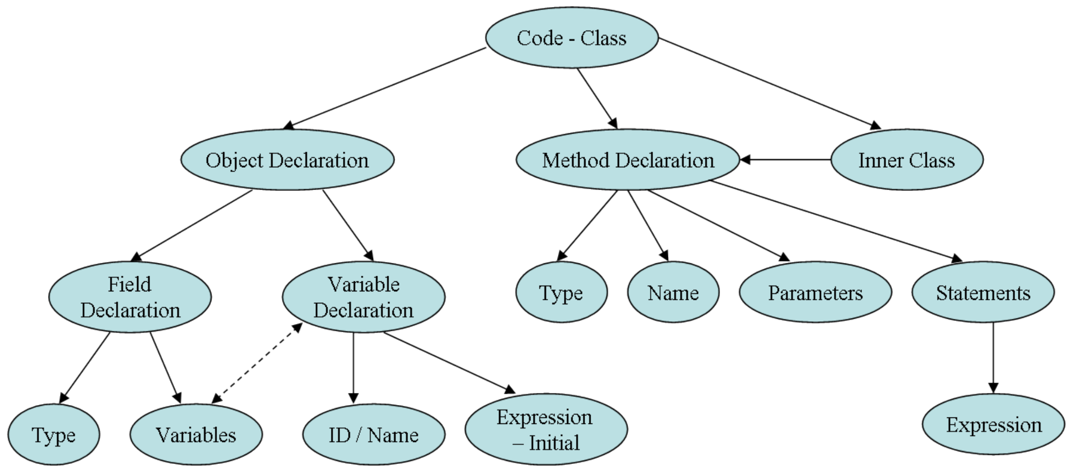


Figure 3: A tree representation of Java source code.

Dynamic analysis has a different problem with losing hidden information. Most dynamic tools only focus on modal windows. A modal window is a GUI window that, once invoked, monopolizes the GUI interaction, restricting the focus of the user to a specific range of events within the window, until the window is explicitly terminated. Hence, the information for the modeless windows in the GUI that do not restrict the user's focus and merely expand the set of GUI events available to the user is not extracted automatically and needs to be added manually. Alternative information about the windows with security requirements is hard to extract unless offering the right password because automation cannot predict or respond to this. Also, dynamic analysis is limited with respect to the underlying functionality it can discover, because by interacting with only the GUI, it is not possible to understand all of the underlying system's behaviours or changes in the system states.

In addition, it is impossible in a static way to extract the information about overlapping windows since this must be determined at run time, so it requires dynamic analysis.

The second issue relates to the source code. If the source code does not correctly implement the requirements, static analysis alone is unlikely to help because it is unable to know what the intended outcome was. But we can compare it with the outcome of the dynamic analysis to fix this issue. In dynamic analysis, the windows with security requirements mentioned above can be extracted in a static way, so a combined approach can also be used.

Static analysis only needs the source code. The source code expresses every part of a software system, including the framework library, GUI implementation (i.e. the setup of GUI and event-handler code), and the underlying application functionality. Our proposed reverse engineering process for such an interactive application is shown in Figure 4.

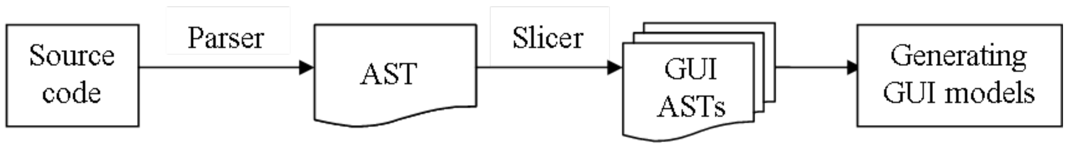


Figure 4: The general process of reverse engineering an interactive system.

We have previously mentioned that we want reverse engineering tools which focus on the user interface as well as on system functionality, i.e. which can handle all parts of an interactive system. Using a Parser, we can collect the information about the GUI widgets, objects, event handlers, and then get the respective behaviour information using a Slicer. We want to understand how we can gather information on GUI structure and objects and internal system functionality respectively to build PModels and PIMs.

In this project, we focus on static analysis and possible ways to achieve it, which are slicing and parsing. The choice of these as the possible ways will be justified in Chapter 2 (after related work). Using parsing, we can abstract the whole application, but by using slicing to just get the parts we are interested in, we can avoid building a huge model. Some problems may occur, such as losing some information like hidden information (i.e. internal system information) during the slicing process, e.g. the count of how many times clicking happened on a button. We propose a possible solution to combine slicing with parsing, and iterate slicing to get each part we are interested in and then emerge with a complete GUI AST for generating models. In this way, we can generate smaller but complete models for an interactive system.

1.3 Report Outline

In the next chapter, we discuss some related works and explain how these were used as the basis for our decision to focus on parsing and slicing. Chapter 3 introduces a small and simple Java/Swing application which will be used as a running example in this project. Then we give an overview of the models (PModels and PIMs) with an example in Chapter 4. In Chapter 5, we present our solution - program slicing, how it works with the example application, and the combination of slicing and parsing. We go on to analyze slicing and parsing in Chapter 6 before we conclude in Chapter 7.

Chapter 2

Related Works

In this section I discuss several papers related to reverse engineering techniques. These papers cover both static and dynamic analysis techniques and tools for a variety of different languages and implementations.

There are many case studies of the use of static analysis techniques, involving model checking. Corbett et al. [6] applied reverse engineering techniques to Java applications. They showed a tool for model checking Java source code called Bandera that automatically extracts finite-state models in the input language of one of the following existing verification tools (e.g. SPIN, SMV, or SAL). The tool has four components, which are the Bandera abstraction engine with a language for specifying abstraction, a slicer to remove irrelevant code in the program for checking an interesting property, a back-end to generate a model with one of the model checker input languages (SPIN, SMV, or SAL), and a user interface for interacting with these components. Bandera uses a slicer to reduce

Java source code based on the user's specification (by describing the semantic features of the program that the user is interested in reasoning about), then abstracts them to get the sliced program for the back-end component to generate a finite-state model in one of the languages and enables translation of the model in the other two model checker input languages.

Dwyer et al. [19] also presented an approach for verifying the specification of GUI internal behaviours using abstraction and then extracting a state-transition model with the SMV model checking tool. Dwyer et al. targeted GUI implementations which are constructed by frameworks or toolkits. Both Dewyer and Corbett's works generate a model for use with an existing model checking tool. The model generated by Dwyer et al. specifically focused on GUI functionality; by contrast, Corbett et al. generated a model focused on specific properties.

The methods described above focus on building models of the implementations. The particular set of models used in our project can also be derived from the implementations but in addition can be derived from the design artifacts as well. The particular models we use in our project are described using the μCharts language to abstract an application with states and transitions. One model, the presentation model, formally models an informal design artifact / implementation by capturing static properties of a UI design, and another model named PIM uses this to capture dynamic UI behaviours to describe the functionality of the design artifact / implementation. We describe these models in detail in Chapter 4. So, our models model an implementation with the structure of its UI and its internal behaviours. Compared to the models generated by the methods discussed above, the particular models used in our project are very different. Dwyer et al. give a clear concept about abstraction in reverse engineering techniques, and the idea about using a slicer is very interesting and suggested a possible approach for our work.

Silva et al. ([14], [15], and [18]) explored the applicability of slicing techniques to the needs of reverse engineering and developed a language-independent approach for reverse engineering interactive systems. They took a Java/Swing application as an example case and reverse engineered a behavioural model directly from the application's source code. They used a parser to build the AST of the application, and then used GUI Code Slicing ([25], [26]) to traverse the AST in order to isolate the Swing sub-program from the entire Java program. Finally they automatically generated a GUI model using the GUISurfer tool [17]. The GUI Code Slicing combines two language-independent techniques including strategic programming and program slicing.

Corbett's Bandera tool contains a slicing component to compress paths in a program by removing some irrelevant properties for checking a property of interest. This similar function is also used by Silva et al. to extract the information related to the GUI layer. Silva et al. produced interesting results by using slicing techniques for reverse engineering graphical user interfaces. Those papers suggested a solution for our project, which is language-independent programming slicing. After parsing, the AST contains all parts of an application. This led me to consider if I could extract anything of interest for building models from the AST using slicing techniques iteratively. This led to more research on slicing techniques ([22], [23], [24], [25], and [26]) which described the history of slicing techniques and many slicing methods, and I applied my research result in this project (see details in Chapter 5).

Another static analysis approach used to reason about user-interaction properties of Swing applications is given in this work [12]. The user-interaction properties refer to the interaction orderings that are the sequences of user interactions that a GUI implementation allows. This paper focused on event-handling, and took Swing applications as targets. It described a notion of modeling event-handling for Swing applications, which helps to explain the structure of event-handling of an example application written in Java/Swing. However, event-handling is not a central concern of our work.

Staiger [16] described static analyses for reverse engineering GUIs for GUI applications, which means they reverse engineered the structure of user interfaces for GUI software systems without their system functionalities whereas for our work we require both. Staiger took C/C++ programs using the GTK/Qt GUI library as an example, collecting the information about GUI elements and their relationships. Staiger used data-flow information to identify the GUI widgets to which these expressions refer, and we can similarly use data-flow information to help building a system dependence graph of an interactive system for the slicing process (see details in Chapter 5).

Static analysis works on the source code of a program and source code expresses all parts of the program. In contrast, dynamic analysis extracts information from the running application meaning the system information is hard to get and can be missed during the analysis as information relating to the system state is often not visible via the GUI. Source code, therefore, provides a better option than the running program for collecting all of the required information. Hence, we prefer static analysis as a reverse engineering technique for our project.

Even though our project focuses on static analysis methods for reverse engineering interactive systems, we still consider dynamic analysis to see if it is relevant to our work. On the dynamic analysis side, Systa [27] focused on the functionality of Java software and used reverse engineering techniques to study and analyse the run-time behaviour of the software. Systa ran the target software under a debugger, and got the event trace information. The event trace information can be viewed as a scenario and is an input to a prototype tool. The prototype tool outputs state diagrams which are used to examine the system's behaviours. This work concentrates on a system's underlying behaviours, ignoring the structure of the UI of the targeted software. Using a debugger to get event traces is a good idea and we consider this useful for future work.

Some of the case studies in dynamic analysis are on testing used in order to detect defects in GUIs. In contrast to Systa's work above, the GUI Ripper tool [7] is a good example of studying reverse engineering of GUIs. It dynamically constructs a model of an executable GUI to help generate test cases under the guide of "test coverage criteria" [20]. A GUI's state is modelled as a set of widgets (GUI objects), properties and values. As well as modelling the structure of a GUI, it also models the GUI's execution behaviour using event-flow graphs. It is similar to the research done by Gimblett and Thimbleby [9], who produced a UI model discovery method to automatically discover a model from a running application. They use the method to firstly explore an interactive system's state space and then use an event flow graph to simulate the actions of a user. However, GUI Ripper requires human intervention since sometimes some windows are missing in the ripping process. Scheetz et al. [28] created a class diagram to represent a system under test and derived test objectives from the class diagram to generate test case with an AI planning system. However the approach used in this paper is not relevant to our research, so we do not consider it for our future work.

One aim of GUI testing is to find inconsistencies and usability problems before the user interface is developed, which can save time and money and is more efficient than finding and fixing those problems after development. In the paper [10], Paiva et al. presented an automatically generated GUI formal model in Spec#, and then automated GUI testing in order to verify the conformity between an implementation and its specification. Spec# is a rich pre/post specification language. Paiva et al. [13] constructed a state machine model of the GUI in Spect# language and mapped information between the model and the implementation. They used a reverse engineering tool to extract structural and behavioural information about the GUI under test, mixing automatic exploration and manual exploration. Compared with paper [10], we view the latter as a newer version of the earlier approach and the latter reduced the manual work required.

The tool called Spec Explorer used in the paper [10] generates test cases with two steps, generating a FSM from a Spec# (specification) and then generating test cases with coverage criteria from the FSM. It led me to consider test case generation of our models for testing purpose in future. PIM is a model described by μCharts (which have a similar structure to FSM), and so we considered the difference between generating test cases from a PIM rather than an FSM. The idea of the mapping tool is similar to another model used in our project named PMR for short (see Section 4.4). PMR presents the relation between a presentation model (PModel) of a system and the system's formal specification. The purpose of building a PMR of a system is to ensure its implementation and functional specification are consistent before development.

There is an increasing demand for transforming user interfaces into a new version, on a different platform from which they are originally implemented. The World-Wide-Web in particular is becoming a target platform as it is the most common interaction environment. Samir et al. [8] developed a dynamic analysis method to automatically migrate Java/Swing applications to Ajax-enabled web-based applications. This approach extracted the structure and behaviour of Java Swing GUIs, using the Aspect tool in the Java application. From the extracted model, it automatically built an Ajax-enabled web application. This work is a meaningful for studying black-box user-interface modernization techniques. These techniques involve extracting an HTML file of the top-level of the GUI application with the purpose of running the instance of the application on the web server, and then sending the HTML file to the client browser. Once the user changes the data or performs an action on the web browser, that change / event would be sent to the server side. At the server side, the original application will be updated with those changes of the user interface, and then the current window on the web will be changed. These methods have some shortcomings, such as having to reload the whole web page to update the changes on the web page. These shortcomings are addressed in the paper [8], which suggests instead only reloading the changes on the web page rather than reloading the whole web page. This paper is not very related to our project, but it is very helpful on migration of GUI applications.

Someone else who did research about migration is Bandelloni et al. [11] who presented the ReverseAllUIs environment to support reverse engineering of user interfaces for different platforms and modalities. They built the corresponding logical descriptions at different abstraction levels. They focused on transformation of web application (XHTML/CSS) to desktop GUI-application, while Samir et al. migrated a GUI application to a web application. Our project targets interactive systems for reverse engineering, whereas this paper targeted web applications, which is unrelated with our project but is a useful technique to migrate a web application to as mobile GUI applications.

Apart from migration between different platforms, reverse engineering techniques are also used to update legacy systems. Moore [21] described the experience with manually static reverse engineering legacy applications to build a model of the user interface functionality. He developed a technique to partially automate the reverse engineering process. The results showed that a language-independent set of rules can be used to detect user interface components from legacy code, and listed some problems that require dynamic analysis to solve. A similar slicing approach was used in his work to identify the user interface subset, including all routines and data structures which are affected by user I/O, and then user interface components from the subset were detected. Moore’s work focused on text-based applications and so is not very useful for our project as nowadays interactive systems have a variety of user interface widgets such as buttons, texts, labels, and menus and so on. However, this paper gave us a clear idea that reverse engineering techniques can be used to update legacy systems.

From the work we have considered above, we have found that whilst there were no existing techniques for performing the sort of reverse engineering of interactive systems we require, there were several ideas which we considered useful for our work. These included the use of programming slicing techniques as a possible solution and test case generation techniques for testing purposes in future.

Chapter 3

Explanation of Example

A simple application I programmed in the Java language using the Swing GUI library is taken as an example interactive system throughout the project. While this is a small application with limited functionality and a small UI, it contains all of the necessary elements to explain my work in the rest of this dissertation while remaining small enough to be easily explained and understood. This game, called the *Guess Game*, asks a player to guess a secret number that the system generates, in a range between 0 and 1000, inclusive of 0 but exclusive of 1000. The system generates randomly a secret number in this range, and then gets the number the player enters. After comparing these two numbers, it opens a message window to tell players that their number guess is higher, lower, or the same as the system's secret number, and provides two options to players to either exit the game or continue to guess until they get the secret number. The goal of the game is to guess the secret number in as few turns as possible. The full source code in Java is given in *Appendix A*.

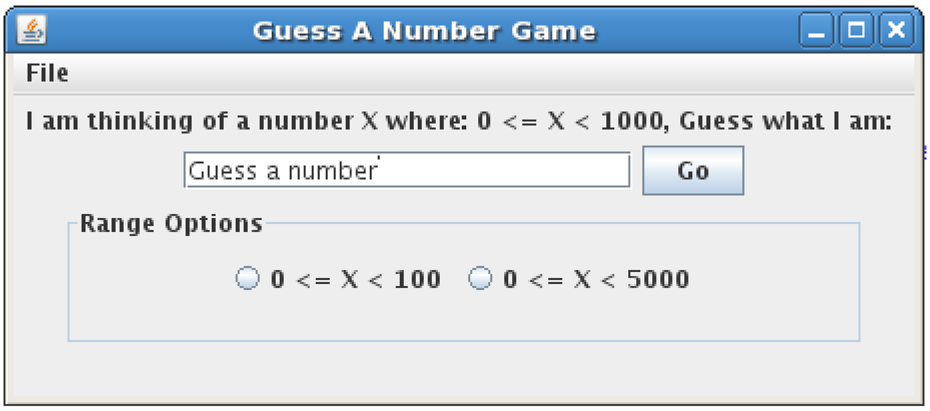
An example of running the Guess Game is as follows:

The initial GUI window named mainWin is the main window of the program. The default range for the number for the players to guess is between 0 and 999. The initial window is shown in Figure 5.

Guessing 700 and typing it in the text field, and then clicking the "Go" button to check leads to message window named msgWin to open and be active (the title bar of the active window is blue). See Figure 6.

The number of 700 is higher, so continue the game by clicking the "Continue Game" button. The message window is closed and the main window is activated. See Figure 7.

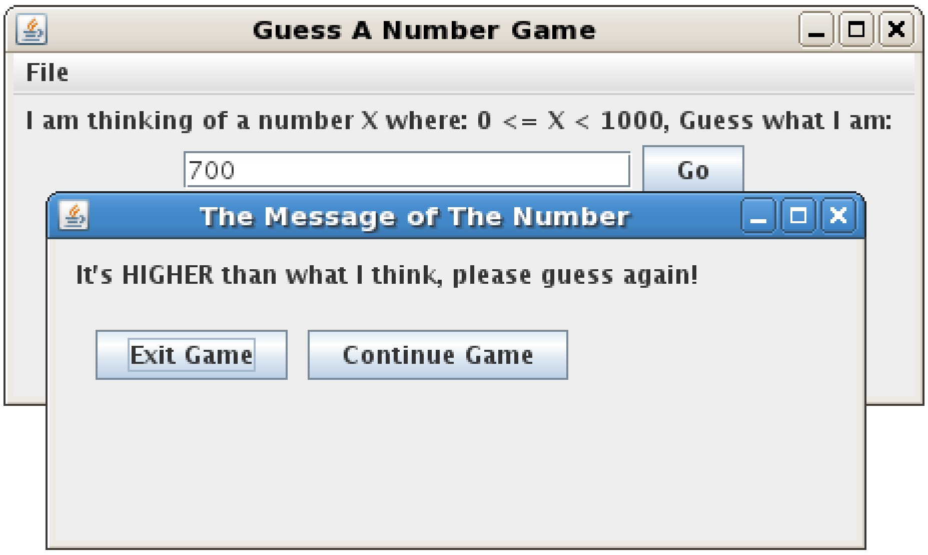
Try 250 and check, see Figure 8.



I am thinking of a number X where: 0 <= X < 1000, Guess what I am:

Figure 5: The main window of Guess Game

The number of 250 is lower, so continue the game and try another number …. This continues until the user enters 262 (which in this example is the secret number) in the text field and checks, as shown in Figure 9.

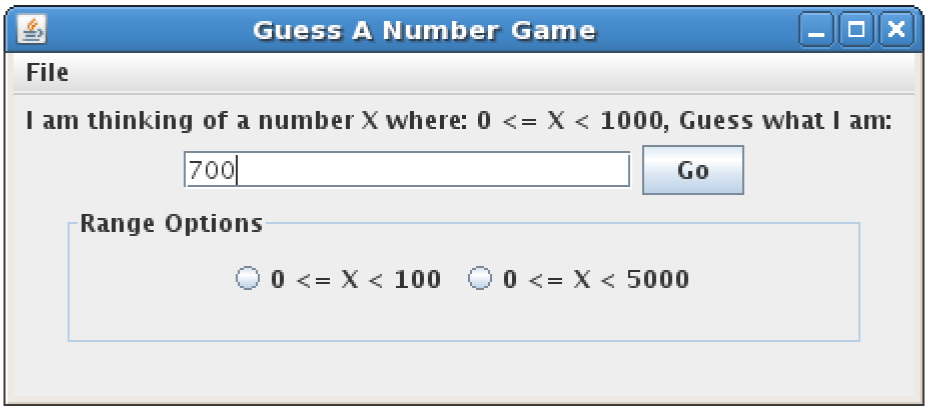


I am thinking of a number X where: 0 <= X < 1000, Guess what I am:

700

It's HIGHER than what I think, please guess again!

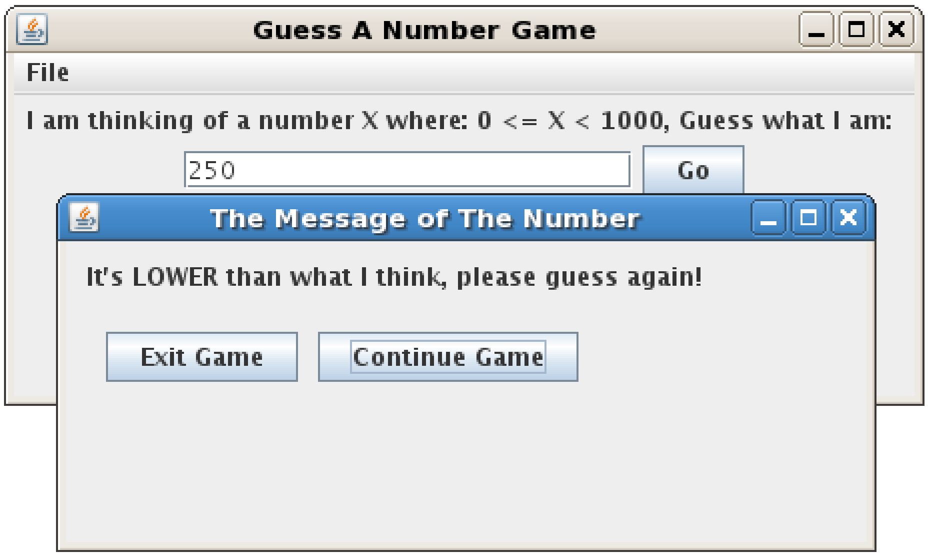
Figure 6: A message window of Guess Game after "Go" button is clicked.



I am thinking of a number X where: 0 <= X < 1000, Guess what I am:

700

Figure 7: The main window after unsuccessful guess.


Guess A Number Game

File

I am thinking of a number X where: 0 <= X < 1000, Guess what I am:

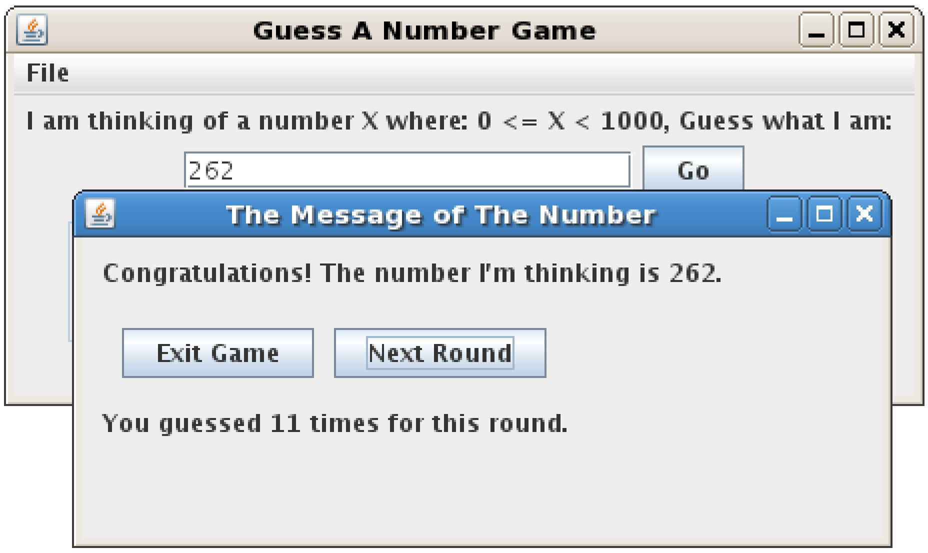
250 Go

The Message of The Number

It's LOWER than what I think, please guess again!

Exit Game Continue Game

Figure 8: The message window after second guess.


Guess A Number Game

File

I am thinking of a number X where: 0 <= X < 1000, Guess what I am:

262 Go

The Message of The Number

Congratulations! The number I'm thinking is 262.

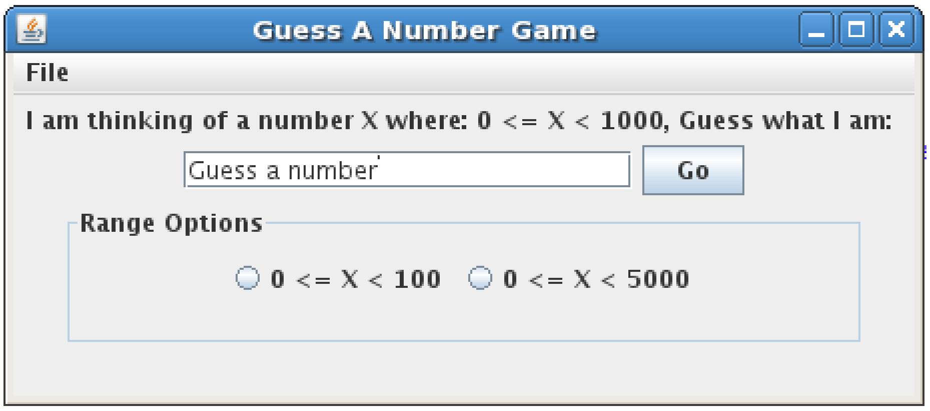
Exit Game Next Round

You guessed 11 times for this round.

Figure 9: The message window after successful guess.

Congratulations! 262 is the secret number. In the back-end of the system, for each time of guessing, it records how many times the player has guessed so far for a round. The process of counting is hidden information and unavailable for players to see from the user interface of the program, but it shows the number of times on the user interface when the player guesses the right number. In this example, the user has tried 11 times to get the correct number of 262. Now the player can either finish the game or start another guessing trip.

Start the next round of the game by clicking "Next Round" button. The message window is closed and the main window is initialized. The counter restarts. See Figure 10.


Guess A Number Game

File

I am thinking of a number X where: 0 <= X < 1000, Guess what I am:

Guess a number

Go

Range Options

0 <= X < 100

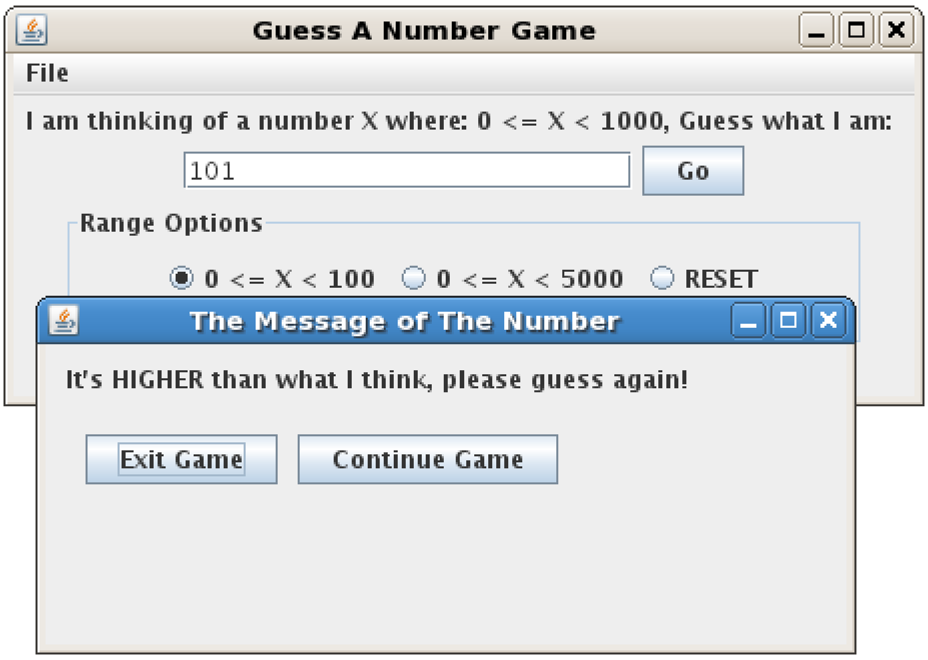
0 <= X < 5000

Figure 10: The main window at start of round two.

In the body of the main window, there are range options to allow players to choose a different range for the secret number either between 0 and 100 or between 0 and 5000.

To try another range click the radio button for "0 <= X < 100", if the next guess from the user is 101 and check. The number of 101 must be higher as it is out of the range. See Figure 11.

This is the first time of guessing for this round, and the counter regarding the number of user guesses is recounting in the backend. You can continue this round by following the rules explained prior, or exit the game by clicking “Exit Game” in the message window to close the program.


Guess A Number Game

File

I am thinking of a number X where: 0 <= X < 1000, Guess what I am:

101

Range Options

[✓ 0 <= X < 100] 0 <= X < 5000 RESET

The Message of The Number

It's HIGHER than what I think, please guess again!

Figure 11: The message window after out of range guess.

If you consider the main window in Figure 11, there is one more option called RESET which appears in the range options. The RESET option is available to the players after they have selected a non-default range. It allows players to reset the range back to 0 <= X < 1000.

Choose “Continue Game” button to close the message window and click RESET radio button on the main window. The game resets. See Figure 12.

The RESET option disappears in the range options area, because the current range is the default range of 0 <= X <1000.

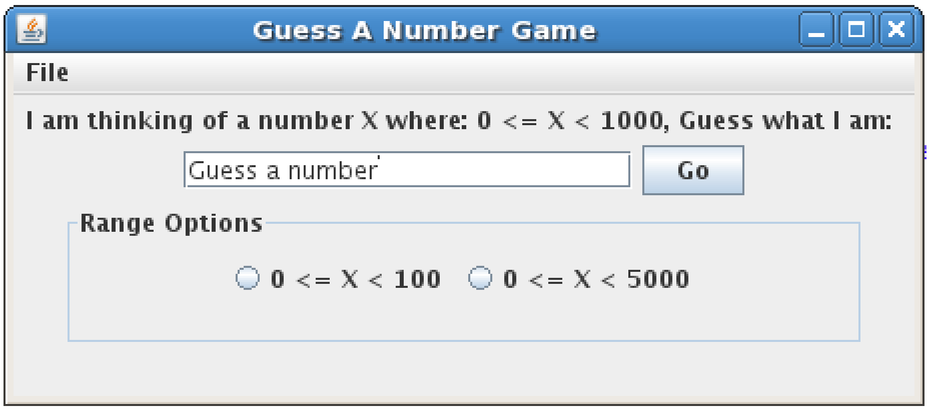


Figure 12: The main window after range is reset.

In the main window, you also can exit the game by clicking the menu *File* and then choosing “Close”. When you click *File*, you will see menu items like in Figure 13.

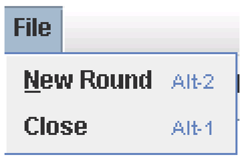


Figure 13: The menu options to end game or start a new round.

The players also can start a new / next round based on the current range option from the menu.

This small example of an interactive system will be used as an example throughout the rest of this report.

Chapter 4

Overview of Models with Example

In this section, I will show how the Guess Game (see Chapter 3 for the explanation of the game) can be described with a set of models which were developed by Bowen and Reeves [4] with the intention of creating formal descriptions of interactive systems.

4.1 Presentation Models

The first model is named the presentation model and is used to formally capture the meaning of an informal design (e.g. a scenario, storyboard or prototype) of a UI or an implemented UI. This kind of model is simple and easy to understand for non-technical people.

This model will formally describe the user interface (or design) of an interactive system by way of its behaviours, and its components (widgets). If used within the design process, the presentation model does not replace the informal design but it is an intermediary between the informal design and formal design process, since it abstracts the understanding of the informal design artifact in a formal way.

> "*The formal structure of the presentation model gives us a different view of the design and enables us to consider it within our formal framework*" [4].

Firstly, we should know the vocabularies of the presentation model: PModel, Widgetname, Category, and Behaviour. *PModel* can be understood as the states of the UI or windows of the program. Each window is described within a PModel which is by way of a set of widget descriptions which are expressed in a tuple consisting of a name, a category, and a set of behaviours. *Widgetname* is a list of names of widgets in each window. *Category* refers to the description of widget categories. *Behaviour* shows what behaviour a widget has associated with it. There are two different types of behaviours. One is an *Interaction behaviour* (indicated by a name prefixed with I_) that affects the UI somehow, opening a different window to the user or making some changes on the current window. The other type is a *System behaviour* (indicated by a name prefixed with S_) that affects the underlying system.

The model begins with a set of declarations which give all of the elements which will be used in the model. An example of the declarations of the Guess Game is given in Table 1.

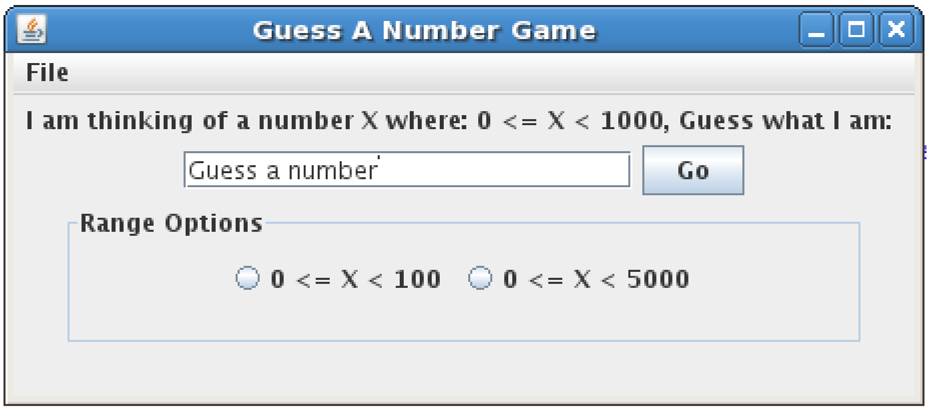
The presentation model for the game is given in Table 2.

In the last row of Table 2, the **:** operator acts as a composition, so that mainWin **:** msgWin consists of all of the widget descriptions of mainWin composed with those of msgWin.

The presentation model GuessGame is the combination of all of the widgets and indicates the total possible behaviours of the user interface. So far, we have an abstract concept for the Guess Game's user interface and its behaviours.

PModel	mainWindow (see Figure 14 below) msgWindow (see Figure 15 below) GuessGame
Widgetname	label1 closeMenuItem msg continueBtn number go range1 rang2 range3 newRoundMenuItem fileMenu exitBtn countMsg
Category	ActionControl SValueSelector StatusDisplay Entry Container
Behaviour	I_openMsgWin S_quitApp I_hideMsgWin S_checkNumbers S_resetApp

Table 1: The declarations of the presentation models of the example system.


Guess A Number Game

File

I am thinking of a number X where: 0 <= X < 1000, Guess what I am:

Guess a number

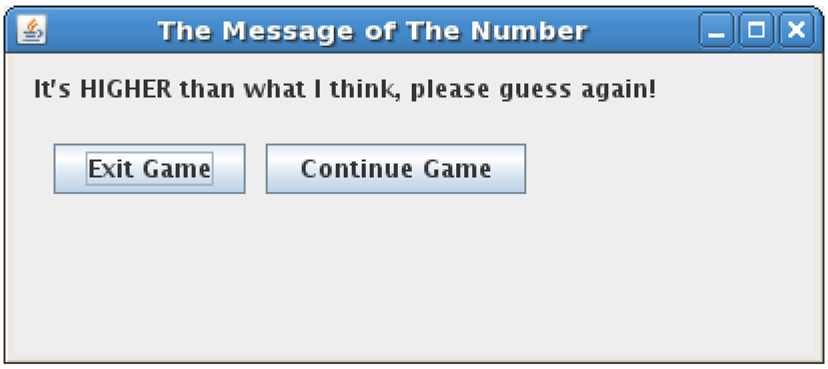
Go

Range Options

0 <= X < 100

0 <= X < 5000

Figure 14: mainWin of the example system.


The Message of The Number

It's HIGHER than what I think, please guess again!

Figure 15: msgWin of the example system.

mainWin is	(label1, StatusDisplay, ()) (number, Entry, ()) (go, ActionControl, (I_openMsgWin, S_checkNumbers)) (fileMenu, Container, ()) (newRoundMenuItem, ActionControl, (S_resetApp, I_hideMsgWin)) (closeMenuItem, ActionControl, (S_quitApp)) (range1, SValueSelector, (S_resetApp, I_hideMsgWin)) (range2, SValueSelector, (S_resetApp, I_hideMsgWin)) (range3, SValueSelector, (S_resetApp, I_hideMsgWin))
msgWin is	(msg, StatusDisplay, ()) (exitBtn, ActionControl, (S_quitApp)) (continueBtn, ActionControl, (S_resetApp, I_hideMsgWin)) (countMsg, StatusDisplay, ())
GuessGame is	mainWin : msgWin

Table 2: The presentation models of the example system.

4.2 Presentation Interaction Models

In the previous section, the presentation model shows the behaviours which exist in the UI, but we also want to ensure that the user can actually reach all of the functionality described in the presentation model. This is the goal of the presentation interaction model (PIM). To show this we must think of how we can let the user move between the windows of the UI. Hence, we need to understand how the UI changes dynamically between the windows of the user interface under the user interactions. The PIM gives us a view of the dynamic changes of the UI. PIM is the composition of the presentation model (PModel) and finite state machines (FSM), described using the μCharts language [38].

The study of FSM and μCharts is beyond the scope of this report, but I will describe them briefly here. The vocabularies of FSM are states and transitions. A *state* refers to a behavioural node of the system in which it is waiting for an event to be triggered. The system is in only one state at a time. The state can change from one state to another when an event is received, that is called a *transition*. The μCharts language has a visual representation, μcharts (the language is μCharts with a capital C, and the visual representations are μcharts with a small c), which also have states and transitions. A discussion of the semantics of μCharts is beyond the scope of this research, but a description of how PIM can be visualized as μCharts can be found in [38]. A μchart consists of a finite set of states, a finite set of action labels which for a PIM are taken from the I_behaviour sets of the PModels, a start state which describes the initial status of the system, an accept state (referred to as a final state), and a transition function which takes a state and an action and returns a state. In addition, a PIM contains a *relation* which relates states to PModels.

The relation between PModels and states of the μcharts is used to link the current active state in PModels and a specific state which the μchart is in. Once there exists a connection, then it represents that this part of the UI described in the presentation model is visible to the user and available for interaction, that shows this part of the UI is reachable by the user. A condition of well-formedness of a PIM is given in [4] as follows:

> "*A PIM of a presentation model is well-formed iff the labels on transitions out of any state are the names of behaviours which exist in the behaviour set of the presentation model which is associated with that state* ."

which means that we can only make a transition between states if an appropriate I_behaviour exists in the PModel related to the starting state of the transition.

Bowen and Reeves use PModels and PIMs to formally capture the information generated by an informal UI design process, and do some things like specify UI behaviours which is useful because it shows that the relevant functionality of a UI are reachable by the user.

The PIM for the example game is shown in Figure 16.

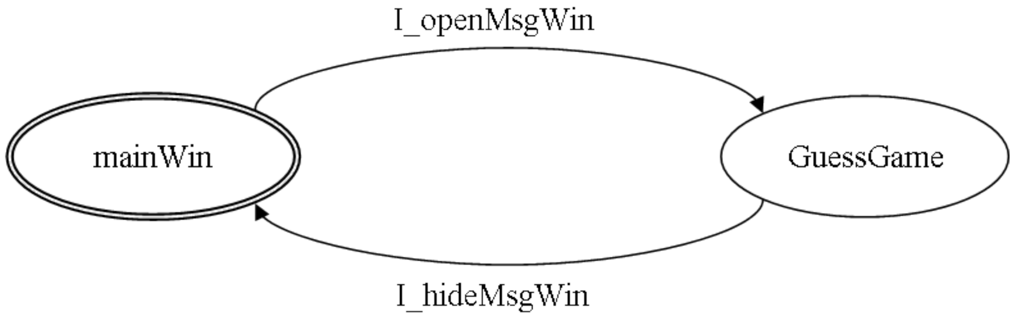


Figure 16: The presentation interaction model of the example system.

The initial window is mainWin which is the initial state. When a user clicks the "Go" button, the program invokes one of the behaviours that is an interaction behaviour – I_openMsgWin to open msgWin. This is indicated in the PModel of mainWin in Table 2 by the widget description:

(go, ActionControl, (I_openMsgWin, S_checkNumbers))

We only consider I_behaviours for the PIM. After the transition, the current state is GuessGame as mainWin and msgWin are both able to seen by the user and both can be interacted with. Once he/she clicks "Continue Game" button on msgWin, it calls the interaction behaviour I_hideMsgWin to hide msgWin (the user is unable to see msgWin) and activates mainWin, so it turns back to the initial state.

In Figure 16, each state is in turn reachable. That means each window of the program is reachable by the users. Even so, it can not guarantee that all behaviours are themselves correct because S_behaviours (system behaviours) described in the presentation model are not visible in the PIM, until we check the PModels for each state. In order to ensure correctness of the functional behaviours of computer systems prior to implementation we can build a functional specification.

4.3 Functional Specification

A functional specification is a formal description of the functional behaviours of a system. It details the behaviours of the system along with properties of inputs and outputs. It has a definite meaning (i.e. fixed semantics) defined in a specification language, such as Z [36]. System operations are specified in the specification by defining how they affect the state of the system. The process of application development can be considerably simplified and streamlined by the specification. The purpose of the functional specification is to show clearly how the varied components of specific applications are to be designed, implemented and integrated with each other. If used correctly, it can substantially save time and cost of application development despite the initial cost of creating and evaluating the specification. This is because it is able to find errors before implementing the system and it is cheaper than finding and fixing errors later in the development process.

To build a formal specification, we typically use the requirements of the system. Here we show how we can give a specification of Guess Game based on a description of its behaviours.

Guess Game (see details in Chapter 3) [1.]randomly generates a secret number in a range for example between 0 and 1000, [2.]compares the number with the input value by the user and then shows the result of the comparison to the user, and the user either chooses to [3.]continue the current round (or start a new round) or [4.]exit the game. Throughout a round, the system [5.]counts for each guess. When the user guesses the secret number, the system [6.]shows a message giving the number of guesses to the user. The system permits the user to either [7.]change a range or [8.]reset a range to start a new round.

The highlighted phrases express the system behaviours of Guess Game, and we will build a functional specification for those behaviours.

The state of a system can be thought of as a collection of values, or observations from the inside of the system, such as the secret number, current guessing count, and current range in our example system. The set of states is called a state space. Hence, the **state space** for the example system has the following observations:

✧ Secret Number

✧ Current Guessing Count

✧ Current Range

Operations of a system act on the system’s state space. The highlighted phrases in the example system’s description above roughly describe the operations. In each operation’s specification, it states the input to the operation and the output from the operation and any changes that are made to any observations. We name each operation in the first row of each item below. We give a functional specification for our example game in natural language below:

1. GenerateRandomNumber
   - Input – no external inputs
   - Changes:

     - Generate a random number within the current defined range (the current range observation).

     - The secret number observation is now updated with the generated number.
   - Output – the generated number
2. CompareNumbers
   - Input – an external input with the user guess
   - Changes:

- Compare the number of the user guess to the secret number observation.

  - The comparison result is either higher, lower or equal.

- Output – the result of these two numbers' comparison

3. ContinueOrNext

   - Input – an external input with one of "continue" or "next"
   - Changes:

     - If the input is "continue", the secret number observation, the current range observation and the current guessing count observation do not change.

     - If the input is "next", initialize the current guessing count observation to be 0, keep the current range observation the same, and generate a new secret number which is used to update the secret number observation.

   - Output – no external outputs

4. CloseGame

   - Input – no external inputs
   - Changes:

     - Exit from the game.

   - Output – no external outputs

5. CountIncrement

   - Input – no external inputs
   - Changes:

- Increase the current guessing count observation by 1.

- Output – no external outputs

6. OutputCountValue

- Input – no external inputs
- Changes – no changes
- Output – an external output with the count value

7. ChangeRange

- Input – an external input with range options
- Changes:

  - Change the current range observation to match the input values

- Output – no external outputs

8. ResetRange

- Input – no external inputs
- Changes:

  - Reset the current range observation to the default value

- Output – no external outputs

From the specified operations shown above, we identify that, there are relations between some of them. We compose the related operation specifications:

◆ CheckAnswer:

2. CompareNumbers

5. CountIncrement

6. OutputCountValue

- ◆ StartNewRound:

3. ContinueOrNext

1. GenerateRandomNumber

7. ChangeRange

8. ResetRange

- ◆ QuitSystem:

4. CloseGame

These operation specifications describe the functional specifications for each system behaviour of the Guess Game. Later I will build a relation between the specifications and the PModels (see Table 2 in Section 4.1). For the purpose of this work, the informal specification given is satisfactory and giving a formal specification developed in *Z* (or any other formal specification language) is beyond the scope of this project.

4.4 Presentation Model Relation

We previously introduced presentation models which give formal meanings to a design artifact (e.g. prototype), and presentation interaction models which add information about the availability and navigation between windows. Those models detail the UI and behaviours. The functional specification is an approach to describe system behaviours. The specification gives the meaning to the system behaviours described in the PModels.

The presentation model relation (PMR) is a relation between presentation models and the functional specification. See the presentation models for our study case developed in Section 4.1, there are three system behaviours, which are S_checkNumbers, S_resetApp, and S_quitApp. The functional specification of the study case is in Section 4.3. The relation between them (i.e. the PMR) is:

S_checkNumbers	→	CheckAnswer
S_resetApp	→	StartNewRound
S_quitApp	→	QuitSystem

So in the PModels if we want to understand the meaning of a behaviour, such as S_checkNumbers, we use the PMR to find out which operation in the specification it is related to. We can then find the details of that behaviour in the specification. This links together the model of the UI with the functional specification.

PIMs describe the interaction behaviours and PMRs ensure the system behaviours and the S_behaviours of the PModels are defined, thus all of the behaviours of the interactive system can be guaranteed to be reached, and we can check correctness and consistency before we start the implementation process.

Chapter 5

Program Slicing

In the previous sections, we have described the particular set of models (the presentation model, the presentation interaction model, and the presentation model relation) we would like to build for interactive software applications. From now on we need to study how we gather the information from the interactive system for building those models. In general, there are two methods to extract the information that we need, dynamic and static analysis. As stated in Chapter 1, this project only focuses on the static technique of reverse engineering. Static methods work with source code.

Figure 4 in Chapter 1.2 gives a general idea of a static analysis process for an application with GUIs. GUI ASTs contain all of the necessary information for generating GUI models. An important role for getting GUI ASTs from the entire AST is played by program slicing, isolating the Swing sub-program from the entire Java program. Slicing techniques are used as an underlying process in many software engineering tools used for different purposes. Slicing can be described as:

> *"a fundamental operation for many software engineering tools, including tools for program understanding, debugging, maintenance, testing, and integration*" [24].

We want to reverse engineer the UI and the system functionality of an interactive software system. To begin, we need to extract GUI ASTs containing the information on the structure of the GUI and underlying system behaviours from an application's source code to build the particular set of models discussed in Chapter 4. For example, the presentation model of the main window of Guess Game (explained in Chapter 3) is shown again in Table3.

In Table 3, the presentation model is constructed from a set of the main window's widgets and its behaviour representation. So, from the source code, we must identify widgets (e.g. the Swing objects for Guess Game which is written in Java using the Swing GUI library) and find out the name, category and behaviour for each of these widgets. Then we subsequently need to categorise the behaviour as an interaction behaviour (prefixed with I_) or a system behaviour (prefixed with S_) by identifying the nature of the behaviour, and finally if it is an S_behaviour find the underlying system code that represents the behaviour by using slicing techniques.

mainWin is	(label1, StatusDisplay, ()) (number, Entry, ()) (go, ActionControl, (I_openMsgWin, S_checkNumbers)) (fileMenu, Container, ()) (newRoundMenuItem, ActionControl, (S_resetApp, I_hideMsgWin)) (closeMenuItem, ActionControl, (S_quitApp)) (range1, SValueSelector, (S_resetApp, I_hideMsgWin)) (range2, SValueSelector, (S_resetApp, I_hideMsgWin)) (range3, SValueSelector, (S_resetApp, I_hideMsgWin))

Table 3: The presentation model of the main window of Guess Game.

Program slicing is the task of computing the parts of a program that directly or indirectly affect the part of a program we are interested in. For example, for the case-study program (the Guess Game in Chapter 3), assume we are interested in the button "Go" of the main window. We can use the program slicing technique to get a sub-program (from the overall program) which only relates to this specified button. In this sub-program, there may be no code about the range options (radio buttons) because they do not affect the values relating to the particular button of interest (the button "Go"). Such a sub-program is referred to as a *program slice*. In the source code, there must be a segment relating to the button "Go" (because the source code describes everything), and we need to specify a point of interest for program slicing. Such a point is referred to as a *slicing criterion*.

5.1 System Dependence Graphs

In general, we abstract a program (e.g. parse a program) and analyse the abstraction of the program (e.g. AST of the program) to draw a graph representation of the program which describes the program's data dependencies and control dependencies. These kinds of graphs include control flow graphs (CFGs) and program dependence graphs (PDGs). In the graph, each vertex represents a statement of the program, and an edge between vertices indicates their control-flow or data-flow. Then we use slicing techniques to find the set of node(s) of the program we are interested in from the CFG or PDG.

The program slices are computed by a backward traversal of the program's control flow graph (CFG) or program dependence graph (PDG) using the slicing criterion to gather the statements and control predicates [25]. For both CFG and PDG, the building blocks are obtained by clarifying data-flow statements and control-flow statements in source code.

Control-flow statements regulate the order in which statements should be executed. These enable a sub-system / program to conditionally execute particular blocks of code. Control-flow statements can be categorized by their effect: decision making statements (e.g. if-else), looping (e.g. for, while), branching (e.g. break, return), and calling. We can see from the source code in Appendix A for the example system that, the most commonly used control-flow statements in the example are decision making statements (if-else) and calling statements. If a sub-system marked as *A* has a calling statement, this control statement passes control to another sub-system marked as *B* but expects to have this control responsibility returned to *A*. For example, an instance of a calling statement from a segment of the Guess Game's source code is as follows, as shown in Segment1.

```
          …
1:        public void NewGame() {
2:            generateRandomNumber();
3:            number.setText("Guess a number");
4:            count = 0;
5:            continueBtn.setText("Continue Game");
6:            msgFrame.setVisible(false);
7:            countMsg.setVisible(false);
8:        }
          …
```

Segment 1: An example of a calling control statement embedded in a sub-system.

In Segment 1, the calling control statement on line 2 belonged to sub-system *A* – `NewGame` from line 1 to line 8, this passes control to another sub-system *B* – `generateRandomNumber` (see code in Appendix A) but the control responsibility of `generateRandomNumber` returns to `NewGame`.

In addition to the control information (described above) embedded in a sub-system, there is also event-based control in our source code. A sub-system is designed to handle an event and then the sub-system responds to the event. For instance, here is a line of code from the source code in Appendix as shown in Segment 2.

```
go.addActionListener(new CheckListener());
```

Segment 2: An example of a calling statement based event.

This code states that an event named `CheckListener` is assigned to a widget named `go`. A sub-system named `public class CheckListener implements ActionListener` handles the designated event. Once this widget is triggered, this system responds to the event by executing the block of code of the sub-system. Event-based control is important for our work because it starts to give us some of the information we need about behaviours of widgets.

Data-flow statements show the flow of processing of the changes of variables in a system, tracing from when data enters the system to where it leaves the system. It is important to be able to identify this as it represents state change in the system. For instance, the value of the `msg` label's text in msgWin from the source code is shown in Segment 3.

Segment 3 shows that, the `msg` label for the Guess Game is declared on line 1, and initialized on line 2. The initial text of the label is empty. Once the system executes the code in the `CheckListener` sub-system, under a different control statement, the text of the `msg` label is changed. This process shows the data flow of the `msg` label in the Guess Game.

Sliced programs are mostly computed by using the *Program Dependence Graph* (*PDG*) representation of a program.

> "*System Dependence Graphs* (*SDGs*) *extend program dependence graphs (PDGs) to incorporate collections of procedures (with procedure calls) rather than just monolithic programs*" [22].

A program's SDG is a collection of PDGs with each PDG for a procedure in the program. More clearly, PDGs are *procedure* dependence graphs rather than program dependence graphs. In general, people use the PDGs for static slicing of single-procedure programs (which refers to programs which only contain one method or inner class), and use SDGs for multi-procedure programs (which refers to programs which have more than one method or inner class). The Guess Game example program is a multi-procedure program, so we use a system dependence graph to represent the program. So, the full process we have devised for reverse engineering our case-study program is shown in Figure 17.

```
             …
1:          Private Jlabel msg;
            …
2:          msg = new Jlabel();
            …
3:          public class CheckListener implements
            ActionListener {
4:             public void actionPerformed(ActionEvent ev){
                  …
5:                try {
                     …
6:                   if (guessNum < RandomNum) {
7:                      msg.setText("It's LOWER than what I
        think, please guess again!");
8:                   } else if (guessNum > RandomNum) {
9:                      msg.setText("It's HIGHER than what
        I think, please guess again!");
10:                  } else {
11:                     msg.setText("Congratulation! The
        number I'm thinking is " + RandomNum + ".");
                        …
                     }
12:               } catch (…) {
                     …
                  }
               }
            }
            …
```

Segment 3: An example of data flow for the `msg` label in Guess Game.

The vertices of the PDG correspond to the individual statements and control predicates of the procedure, and the edges of a PDG correspond to data and control dependencies among the procedure's statements and predicates. Also, a PDG of a procedure is obtained by merging its data dependence graph and control dependence graph. The edges of a PDG define the ordering that the procedure uses when executing statements, which preserves the semantics of the procedure.

A call statement (a calling control statement) is represented by a *call vertex* and a set of *actual-in* and *actual-out* vertices for parameters, as we see later, these may be explicit (in the definition of the procedure) or implicit (i.e. global variables the procedure accesses). For each parameter, there is an actual-in vertex and there might be an actual-out vertex which may be modified during the call.

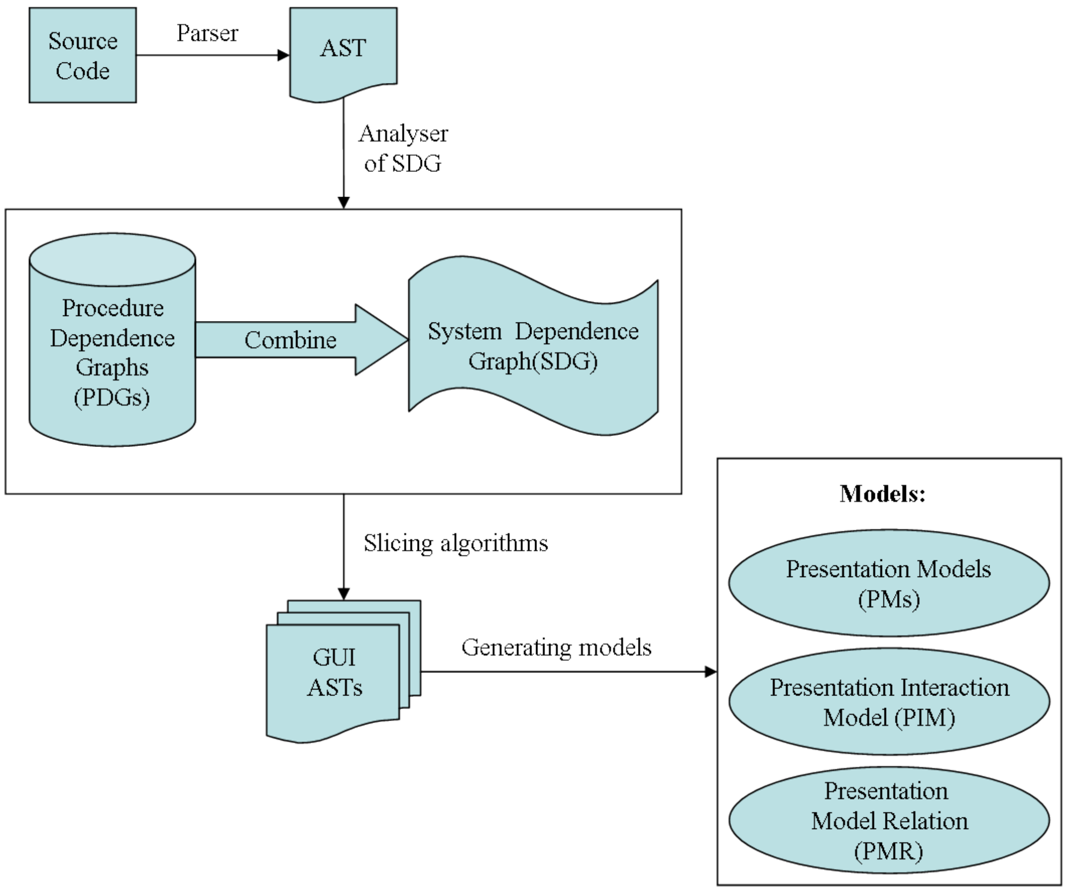

Source Code
Parser
AST
Analyser of SDG
Procedure Dependence Graphs (PDGs)
Combine
System Dependence Graph(SDG)
Slicing algorithms
GUI ASTs
Generating models
Models:
Presentation Models (PMs)
Presentation Interaction Model (PIM)
Presentation Model Relation (PMR)

Figure 17: The actual process of reverse engineering an interactive system which has multiple procedures.

For example, see the source code in Segment 1 above, this can be described in the SDG is shown in Figure 18.

Figure 18 gives the PDG for the sub-system – `NewGame`. The control edges from left to right and top to bottom specify the ordering of the statements to be executed. The called procedure is `generateRandomNumber` without explicit parameters, so there are no actual-in or actual-out vertices for explicit parameters but only a call vertex `Call generateRandomNumber` for the call statement.

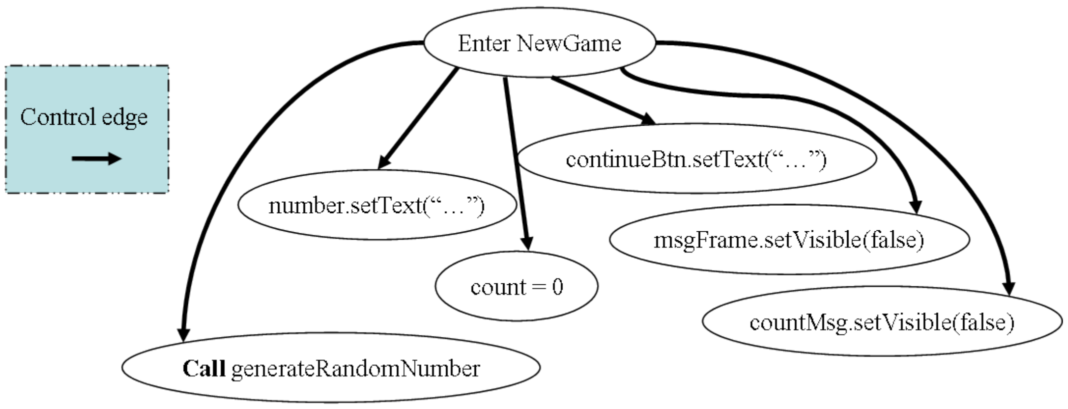

Control edge
Enter NewGame
continueBtn.setText("…")
number.setText("…")
msgFrame.setVisible(false)
count = 0
countMsg.setVisible(false)
Call generateRandomNumber

Figure 18: The partial SDG of Guess Game for the Segment 1.

Recall that we treat global variables as "extra" parameters, which can lead to additional actual-in and actual-out vertices. For the called procedure `generateRandomNumber` in Segment 1, we can collect its related source code from the Guess Game's source code listed in Appendix A and display the collected source code in Segment 4.

Segment 4 shows that this procedure has global variables that are `range1`, `range2`, `range3`, and `RandomNum`. This means that there are extra actual-in vertices and actual-out vertices for the call statement. The PDG of Segment 1 can be completed with global parameters for the calling statement displayed in Figure19.

```
…
private static int RandomNum;
private JRadioButton range1;
private JRadioButton range3;
private JRadioButton range2;
…
public void generateRandomNumber() {
   if (range1.isSelected()) {
      RandomNum = (int)(Math.random()
      *100);
      range3.setVisible(true);
   }
   else if (range2.isSelected()) {
      RandomNum = (int)(Math.random()
      *5000);
      range3.setVisible(true);
   }
   else {
      RandomNum = (int)(Math.random()
      *1000);
      range3.setVisible(false);
   }
}
```

Segment 4: The source code related to the procedure `generateRandomNumber`.

The called procedure has an *entry vertex* and a collection of *formal-in* and *formal-out* vertices. Similarly, as global variables are treated as "extra" parameters, they give rise to additional formal-in and formal-out vertices. For instance, the dependence graph for the called procedure "`generateRandomNumber`" for

Segment 1 is depicted in Figure 20.

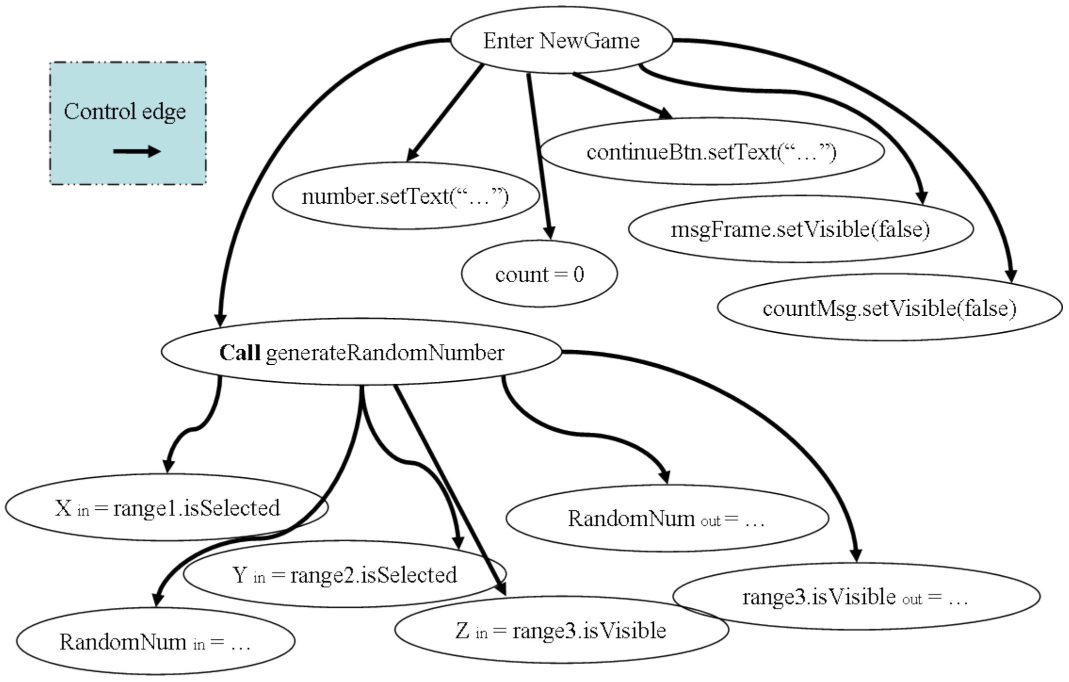
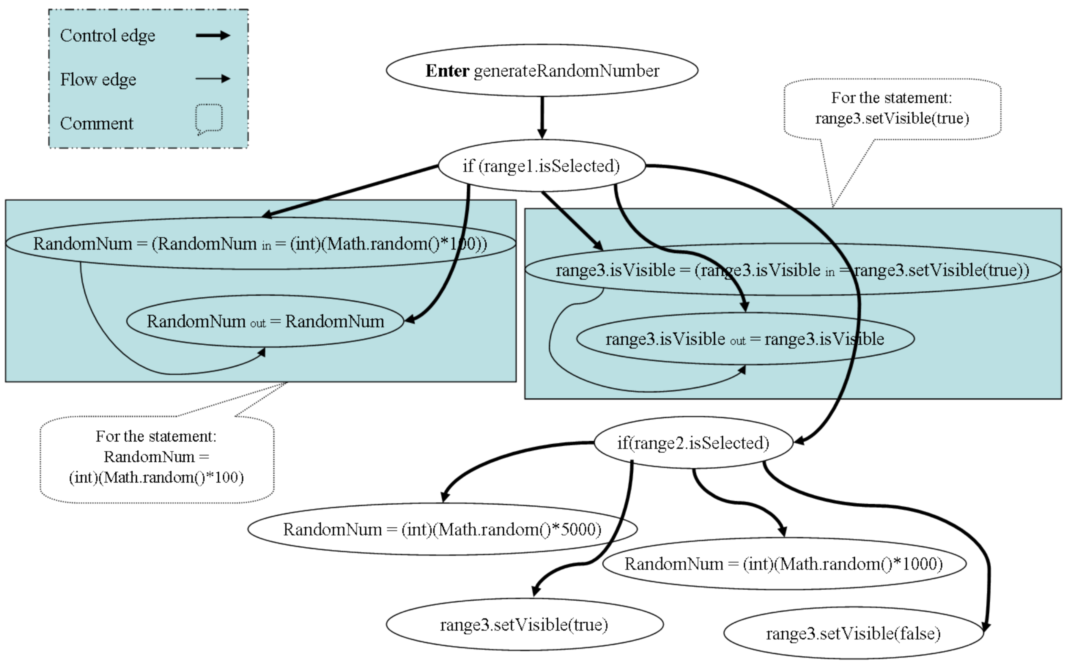


Figure 19: The partial SDG of Guess Game for Segment 1 with additional actual-in and actual-out vertices as global variables in the called procedure.

Figure 20 shows the partial PDG for the called procedure, we show only part here to keep the figure simple and clear to read. We only add formal-in and formal-out vertices for two global variables of two statements on line 7 and line 8 in Segment 4, which depended on the control statement `if (range1.isSelected)` of line 6 in Segment 4. The left shaded part in the PDG represents the statement on line 7 regarding the global variable named `RandomNum`, and the right shaded part represents the statement on line 8 regarding the global variable named `range3`. The rest of the vertices of the PDG can be completed in a similar way as shown by the shaded parts.



Control edge
Flow edge
Comment
Enter generateRandomNumber
For the statement: range3.setVisible(true)
if (range1.isSelected)
RandomNum = (RandomNum in = (int)(Math.random()*100))
RandomNum out = RandomNum
range3.isVisible = (range3.isVisible in = range3.setVisible(true))
range3.isVisible out = range3.isVisible
For the statement: RandomNum = (int)(Math.random()*100)
if(range2.isSelected)
RandomNum = (int)(Math.random()*5000)
RandomNum = (int)(Math.random()*1000)
range3.setVisible(true)
range3.setVisible(false)

Figure 20: The partial PDG of the called procedure `generateRandomNumber` with source code in Segment 4.

The PDGs are connected together to form the SDG of the system by call edges and by parameter-in and parameter-out edges. *Call* edges represent procedure calls and run from a call vertex to an entry vertex. *Parameter-in* and *parameter-out* edges represent parameter passing. A parameter-in edge runs from an actual-in vertex to its corresponding formal-in vertex, and a parameter-out edge runs from a formal-out vertex to an actual-out vertex. Combining the PDG of the called procedure in Figure 20 and the partial SDG in Figure 19 for the combination of Segment 1 and Segment 4, gives the SDG shown in Figure 21.

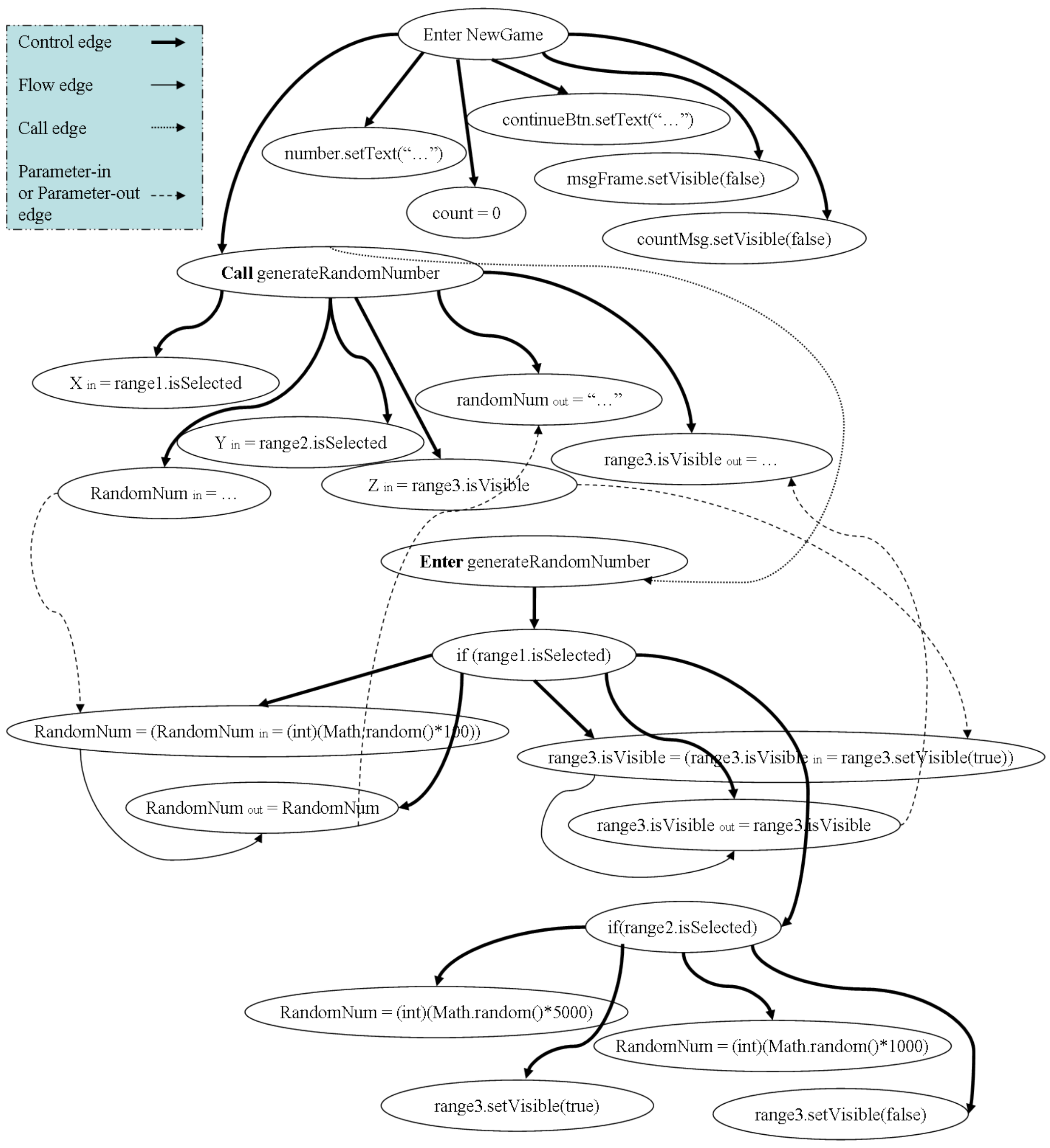

Control edge
Flow edge
Call edge
Parameter-in or Parameter-out edge
Enter NewGame
number.setText("…")
count = 0
continueBtn.setText("…")
msgFrame.setVisible(false)
countMsg.setVisible(false)
Call generateRandomNumber
X in = range1.isSelected
Y in = range2.isSelected
Z in = range3.isVisible
RandomNum in = …
randomNum out = "…"
range3.isVisible out = …
Enter generateRandomNumber
if (range1.isSelected)
RandomNum = (RandomNum in = (int)(Math.random()*100))
RandomNum out = RandomNum
range3.isVisible = (range3.isVisible in = range3.setVisible(true))
range3.isVisible out = range3.isVisible
if(range2.isSelected)
RandomNum = (int)(Math.random()*5000)
RandomNum = (int)(Math.random()*1000)
range3.setVisible(true)
range3.setVisible(false)

Figure 21: The partial SDG of the combination of Segment 1 and Segment 4, adding a call edge, parameter-in edges, and parameter-out edges.

In Figure 18, each node represents a statement of the procedure in Segment 1. As the example system is written in Java using the Swing GUI library, the features and constructs of Swing allow widgets to directly call methods from the library. Hence, the expression of some nodes can be extended. For example, the node of `countMsg.setVisible(false)` can be expressed with more details. The widget named `countMsg` calls a procedure named `setVisible` with the actual-in parameter valued `false` and executes this procedure to set this widget invisible. It means that this node puts the call statement and procedure entry together for the `setVisible` method. If we decompose the statement: `countMsg.setVisible(false)`, we can show it with the SDG representation in Figure 22.

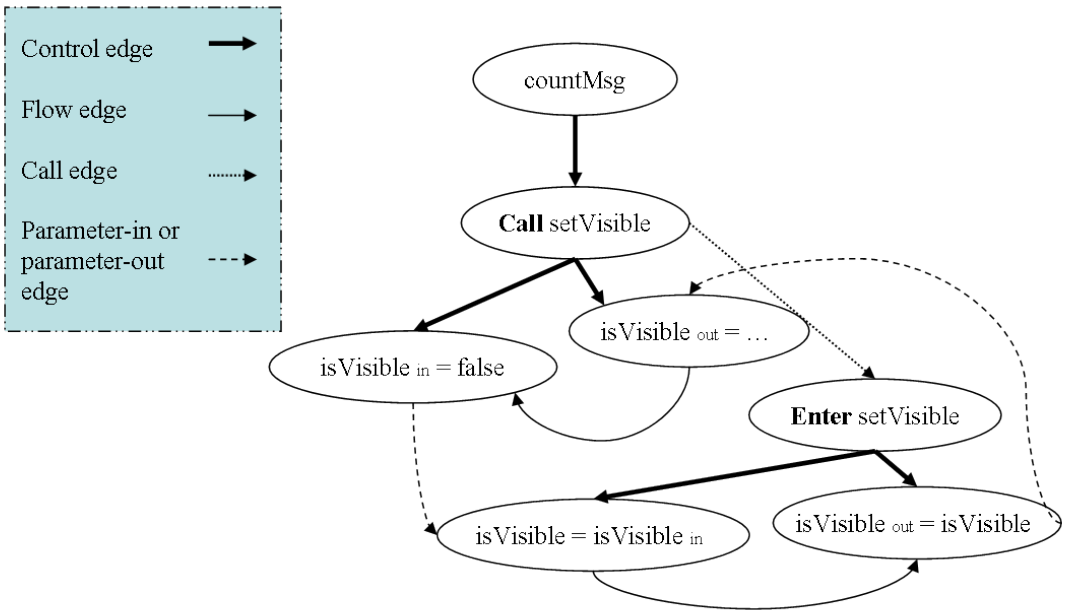


Figure 22: The extension of the node `countMsg.setVisible(false)` of Figure 18.

Firgure 22 shows the complete expression of the node `countMsg.setVisible(false)` of Figure 18, where `setVisible` can be directly used from the Swing library. A similar expression also occurs in Figure 20 and Figure 21, for the `range3.setVisible(true)` statement. In all of the figures used in this work, thick solid arrows represent control dependencies, thin solid arrows correspond to data dependencies, small dashed arrows are used for call edges, and big dashed arrows represent parameter-in or parameter-out dependencies. In the specified statement, there exists parameter passing and output of the actual value of parameters. In order to simplify the SDG, we only build the SDG based on statements and ignore the extension expressions caused by the Swing library for the later examples.

The full SDG of Guess Game is shown in Appendix B. The complete SDG will be very big due to the inclusion of the extra vertices because of global variables and the Swing library property, but we show the main structure of the SDG by omitting those extra vertices but keeping their relationships.

5.2 Slicing of a System

In general, source code of a system can consist of thousands of lines of codes. It is not practical to understand source code manually and it would cost too much time to consider the large amount of source code in this way to build the models. If we do so, we end up with large amounts of information which must be read and understood to build the PModel or PIM, which makes model development slow and resource intensive, as the large amount of information generated from the source code must all be analysed to build the final models. Slicing techniques help to reduce unnecessary source code and get the relevant source code for building models of an application. It saves time if we only have to consider the relevant source code for model generation and is therefore a more efficient process.

Horwitz et al. [24] showed that system slices can be obtained by solving a reachability problem on the SDG. To compute the program slice with the slicing criteria with respect to a SDG vertex *v*, there are existing *realizable* paths from some SDG vertices to *v* along control and data flow edges. The set of outgoing vertices on those paths make up the sliced SDG, and their corresponding statements are composed to create the program slice with respect to the statement of *v*.

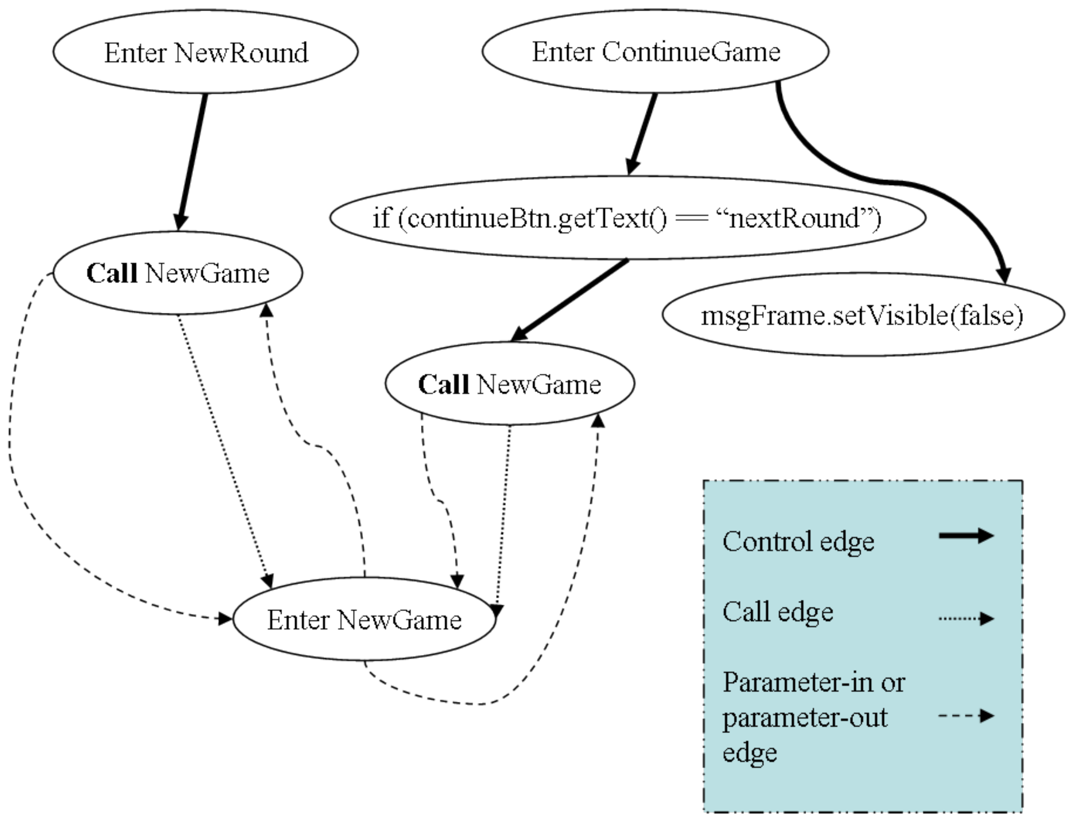
We only consider realizable paths not all paths, because not all paths in the SDG correspond to possible execution paths. For example, we firstly build a SDG for Segment 5, and then check the SDG in Figure 23.

We have explained the treatment of global variables for the SDG in the previous section (see Figure 19, `generateRandomNumber`). Similarly, there are global variables in Figure 23, which are `continueBtn` and `msgFrame` for Segment 5, `number`, `count`, and `countMsg` for the called procedure `NewGame` whose source code is in Segment 1. Hence, there must be extra actual-in vertices representing the value of each actual parameter before the call and actual-out vertices representing the value of each actual parameter after the call `NewGame`.

We omit the PDG for NewGame in this figure. However, the procedure entry of NewGame has both formal-in vertices and formal-out vertices, similar to the procedure entry of generateRandonNumber in Figure 20. In order to keep the figure simple, we ignore those vertices and the detailed PDG for the called procedure but keep the relation between the call vertices and procedure entry vertex in Figure 23 for discussing realizable paths. There are parameter-in edges from actual-in edges to formal-in edges, and parameter-out edges from formal-out edges to actual-out edges. Thus, the relation between the call statements and the procedure entry in Figure 23 include call edges, parameter-in edges and parameter-out edges.

```
 …
public class NewRound implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        NewGame();
    }
 }

 public class ContinueGame implements ActionListener {
    public void actionPerformed(ActionEvent e) {
         if (continueBtn.getText() == "Next Round") {
           NewGame();
        }
        msgFrame.setVisible(false);
    }
 }
```

Segment 5: Two different procedures with the same procedure call.



Enter NewRound
Enter ContinueGame
if (continueBtn.getText() == "nextRound")
Call NewGame
msgFrame.setVisible(false)
Call NewGame
Enter NewGame
Control edge
Call edge
Parameter-in or parameter-out edge

Figure 23: The simplified SDG of Segment 5.

We can consider Figure 23 and Figure 21 together to view the detailed SDG of Segment 5, which describes the detailed SDG of multi-procedures consisting of `NewRound` and `ContinueGame` invoking `NewGame`, and `NewGame` invoking `generateRandomNumber`. In order to discuss realizable paths more clearly with our example, we just focus on the simplified SDG in Figure 23.

To compute a slice with respect to `msgFrame.setVisible(false)`, we only need to consider the realizable paths, without the path: `NewRound -> call NewGame -> NewGame`.

Realizable paths:

> "*reflect the fact that when a procedure call finishes, execution returns to the site of the most recently executed call*" [24].

In a call site, there are outgoing parameter-in edges, incoming parameter-out edges, and outgoing call edges. A path in the SDG is a realizable path if and only if its call edge, parameter-in edges and parameter-out edges work on the same call statement and called procedure. For example, see the path labeled with ① in Figure 24:

```
NewRound -> call NewGame -> enter NewGame -> call
NewGame
```

is a realizable path, while the path labeled with ② in Figure 24:

```
NewRound -> call NewGame -> enter NewGame -> call
NewGame
```

is not. In path ①, the parameter-in edge (`call NewGame -> enter NewGame`) and the parameter-out edge (`enter NewGame -> call NewGame`) work on the same call statement and procedure entry, while the parameter-out edge (`enter NewGame -> call NewGame`) works on a different call statement than the parameter-in edge (`call NewGame -> enter NewGame`) in path ②.

Therefore, using realizable paths with a slicing algorithm, it's possible to get a precise program slice, for a given vertex *v*. The slice is represented in the SDG by the set of vertices that lie on some realizable paths from the entry vertex of the procedure to *v*. To achieve this precision, Horwitz et al. [24] use the augmented SDG with *summary edges*. A summary edge is added from actual-in vertex *v* to actual-out vertex *v* whenever there is a realizable path from the actual-in vertex to the actual-out vertex. Also, the summary edges exist in a call statement and represent the interprocedural (describing between different procedures) data dependencies. As we do not display the extra vertices as the global variables in Figure 23, we take the previous example of the SDG containing the calling statement `generatRandomNumber` with extra vertices in Figure 19, and augment this SDG in Figure 25.

Similar to the example of Figure 23, the summary edges exist between actual-in vertices and corresponding actual-out vertices, but it is hard to explicitly state them in Figure 23 because we ignore the global variables and the extra expressions of some nodes representing methods, as those methods are directly used from the Swing library. However, this concept of the summary edge should be kept in mind while using the SDG with slicing methods to get a precise program slice.

With the augmented SDG, we use two-passes to do the program slicing, and each pass only traverses certain kinds of edges. Pass 1 starts from the slicing criterion – vertex *v*, and backwardly traverses along data-flow edges, control-flow edges, call edges, summary edges, and parameter-in edges, but not parameter-out edges. Pass 2 starts from all of the actual-out vertices reached in Pass 1, and then backwardly traverses along data-flow edges, control-flow edges, summary edges, and parameter-out edges, except call edges or parameter-in edges. The sliced program (program slice) consists of the set of vertices obtained during traversing Pass 1 and Pass 2, and the edges between those vertices.

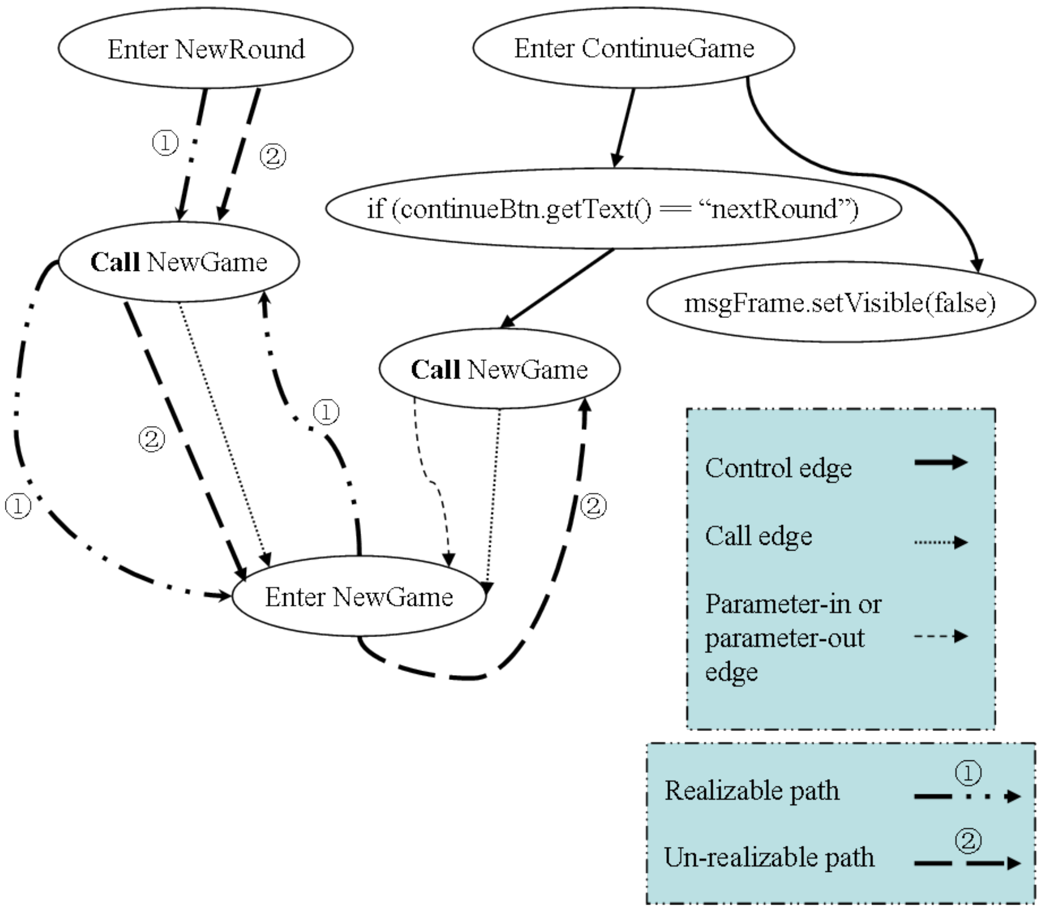


Figure 24: An example of a realizable path in the SDG.

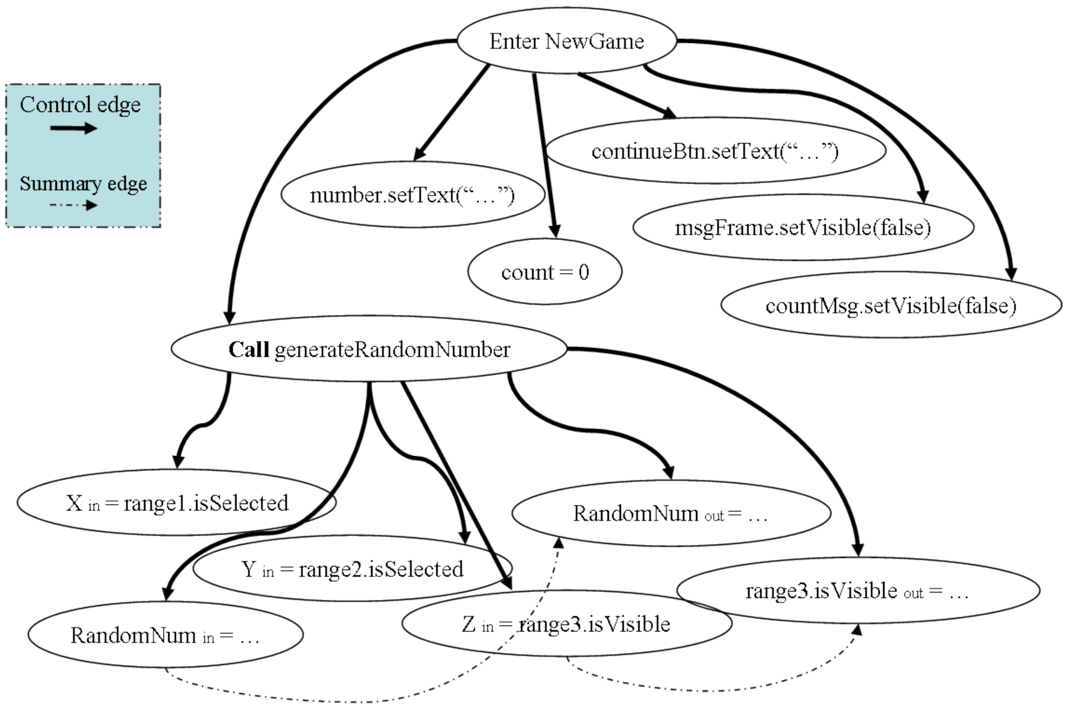


Figure 25: The SDG of Figure 19, augmented with summary edges.

Keeping in mind the summary edges for Figure 23, if we slice it with respect to `msgFrame.setVisible(false)` by the two-passes method, we can show the vertices and edges traversed by the method in Figure 26. The result of an interprocedural slice of Figure 26 is shown in Figure 27.

In short, the two-passes traversal for the slicing algorithm can be described as follows: Pass 1 determines all vertices from which a vertex *v* of interest can be reached without traversing procedure entries. Procedure entries can be ignored in Pass 1, as the summary edges guarantee the data dependencies between multi-procedures. Pass 2 determines the remaining vertices in the slice by traversing all of the procedure entries omitted in Pass 1.

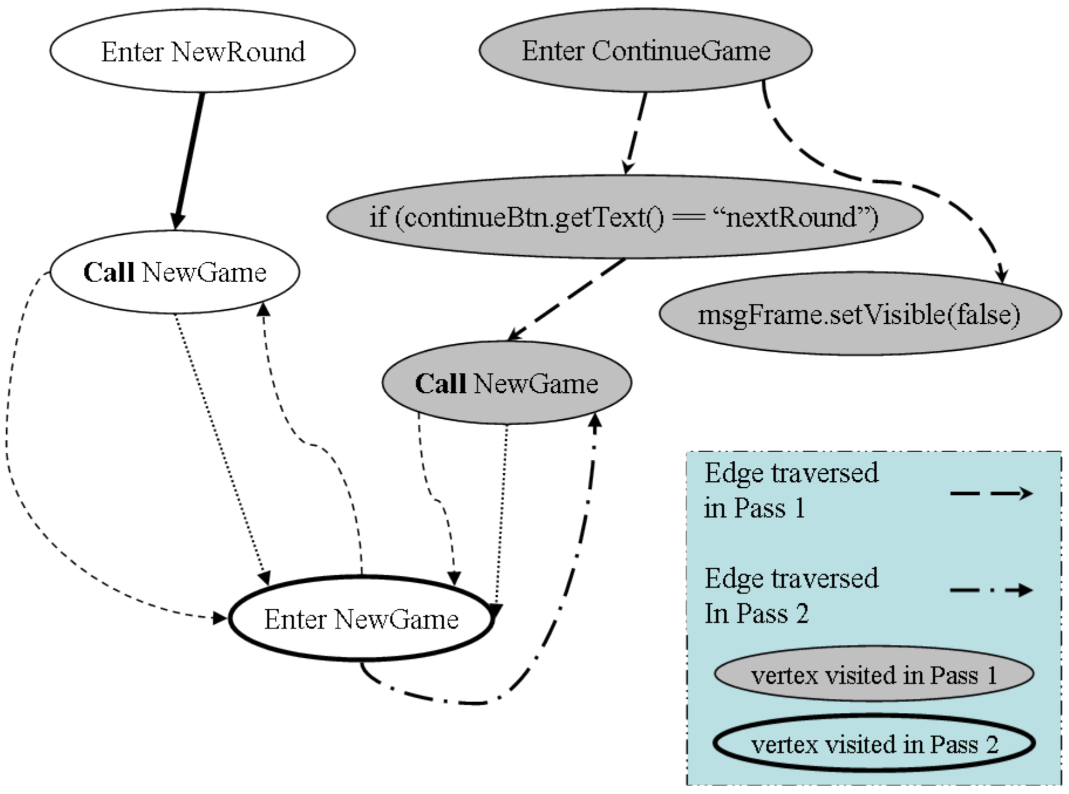


Figure 26: The SDG of Figure 23, sliced with respect to `msgFrame.setVisible(false)`.

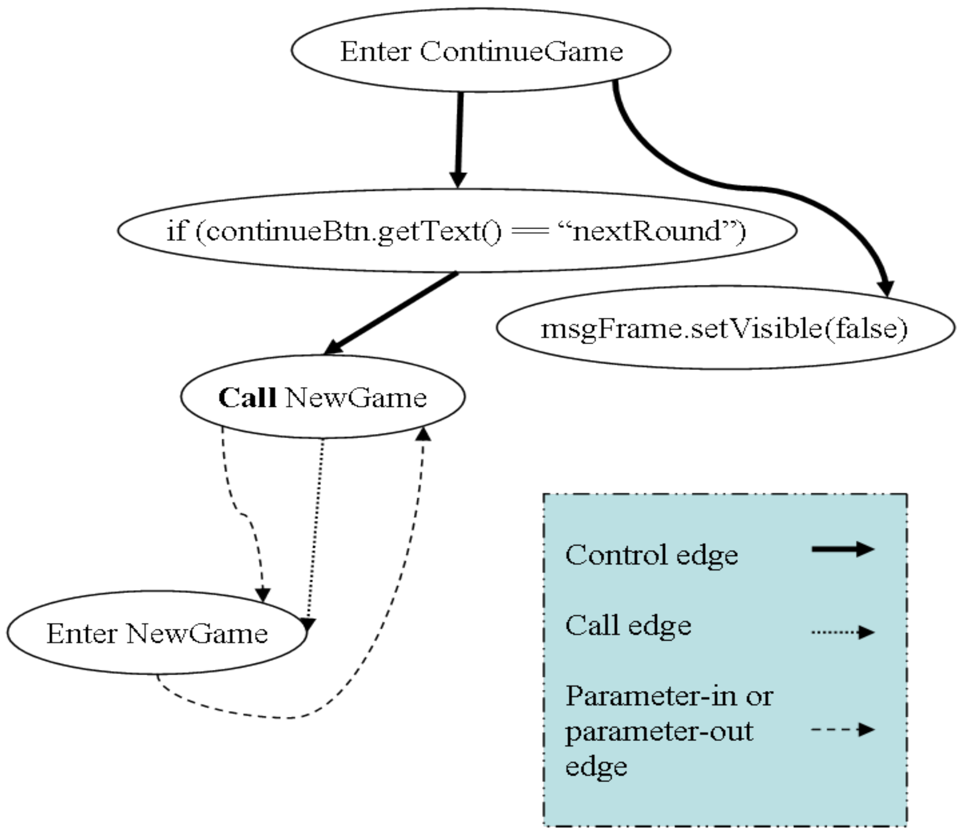

Enter ContinueGame
if (continueBtn.getText() == “nextRound”)
msgFrame.setVisible(false)
Call NewGame
Enter NewGame
Control edge
Call edge
Parameter-in or parameter-out edge

Figure 27: The sliced SDG of Figure 26.

5.3 Slicing with JavaParser

The example SDGs presented in the previous section are manually obtained from source code. It is not too hard to get those SDGs manually because the source code for the case-study system (the Guess Game application) is not too long. If we work on an interactive system for example with more than 2000 lines of source code (and it is common for interactive systems to have far more lines of code than that), it will be quite hard to picture the SDG for the whole system, unless there is a tool to automatically analyze the source code and then generate the SDG. We

can envisage such a tool using Parsing tools, such as Java Parser which can be obtained from JavaParser's sourceforge page for Java applications:
http://code.google.com/p/javaparser/

We start by using JavaParser to generate the AST (abstract syntax tree) of a system, which models an entire representation of the abstract syntactic structure of source code in a tree form. Figure 3 in Chapter 1 gives a tree representation of some Java source code. Each node in Figure 3 can be extracted by the parsing technique. The AST contains all of the information about the system. Using JavaPaser, we can get the information required for the SDG construction by traversing the tree. The output of the Guess Game with JavaParser is displayed in Appendix C. Now we have the AST we can build the SDG manually. This is better than manually building the SDG directly from the source code, but is still not very efficient. There is an existing commercial program-understanding tool called CodeSurfer (see http://www.grammatech.com/products/codesurfer/) for the C++ programming language which creates Call graphs and does dataflow analysis and control dependence analysis and can construct the PDGs or SDGs for systems. Such a tool could be developed for the Java programming language to automatically build the SDGs for Java applications in the same way. In the absence of a tool to build the SDG for our application we proceed manually. The process we describe in this thesis which shows how to build the SDG from an AST could be used as the basis for developing the sort of tool described above (see Chapter 7 future work).

In the SDG, the first ENTER vertex is always named by the system's name from the super-class. For example, in our Guess Game, the first Enter vertex to start the SDG is `ENTER guessGame`, we set the name from the class declared in the top node under the import libraries as shown in Segment 6.

The class listed in Segment 6 is the super class, representing the system. After getting the first Enter vertex, we clarify the control dependencies to build vertices for each control statement and add an arrow from the Enter vertex to those vertices. The control statements include object declarations, method declarations and inner class declarations. JavaParser is able to get the global variables with object declarations, each method with its name, parameter information and body statements, and any inner class information. The name of the method is used to construct the Enter vertex of the PDG of the method procedure.

For example, if we used JavaParser to parse segment 6, then the output would be as shown in Table 4. (Appendix C gives the actual parser outputs.)

Then according to the output in Table 4, we are able to generate the SDG of Segment 6. Each super-class has one `main` method for the system. When the program is run it is the `main` method which subsequently calls the other procedures. Hence, there is no call vertex for the `main` method only the Enter vertex. The first level of the control-flow information for the `guessGame` class includes object declarations and the `main` method. We build the SDG for the first level of control-flow as shown in Figure 28.

```
import javax.swing.*;
…
public class guessGame {
    private static int RandomNum;
    private JTextField number;
    private JLabel msg;
    …
    public static void main(String[] args) {
        RandomNum = (int)(Math.random() *1000);

        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new guessGame().createAndShowGUI();
            }
        });
    }

    public void createAndShowGUI() {
         try {
        …
        } catch {
        …
        }
    }
    …
}
```

Segment 6: The partial source code of the class `guessGame` for Guess Game.

Object Declaration:	int RandomNum JTextField number JLabel msg …

Method name:
 	
Type:
 	
Parameter:
 	
Body-
 	
blockstatement:
	Main
	
	void
	
	[String[] args]
	
	{
	    1. RandomNum = (int)(Math.random() *1000);
	    2. SwingUtilities.invokeLater(new Runnable() {
	        public void run() {
	            *new guessGame().createAndShowGUI();
	        }
	    });
	}

Method name:
 	
Type:
 	
Parameter:
 	
Body-
 	
blockstatement:
	createAndShowGUI
	
	void
	
	null
	
	{
	    try {
	        …
	    } catch {
	        …
	    }
	}

…	…

Table 4: Output of parsing Segment 6.

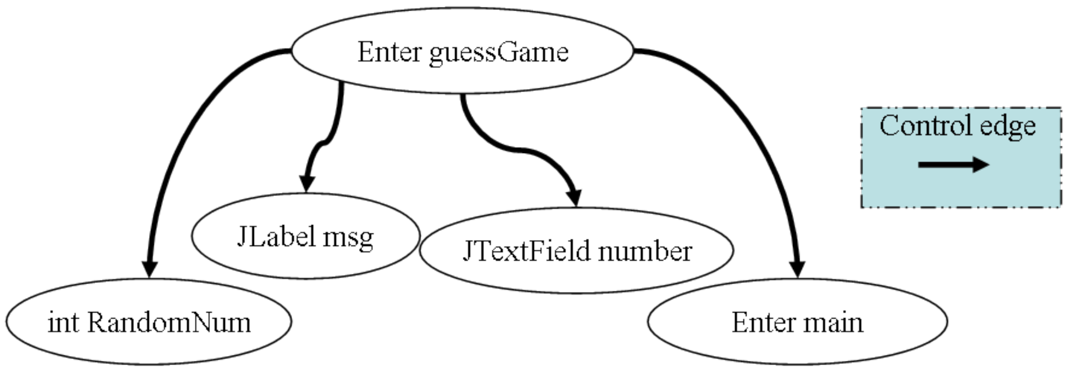


Figure 28: The partial SDG of Segment 6.

In order to complete the SDG in Figure 28, we need to picture the procedural dependence graph (PDG) for the `main` procedure. Similarly, we can build a PDG for each method, or inner class, by analyzing the corresponding block of code obtained by JavaParser. It is straightforward to write methods which obtain the relevant parts of the AST, and JavaParser uses the visitor pattern [2] to make this easier. We use the method's name or the inner class's name to build the ENTER vertex of its PDG respectively, and then analyze its body statements which may contain decision making statements such as if-else to build control vertices. We show the output of the `main` procedure in Table 4. There are two statements containing one event-based calling statement.

```
SwingUtilities.invokeLater(new Runnable() {
    public void run() { … }
})
```

This event-based calling statement is a commonly-used statement for the `main` method for any Java application using the Swing library, and its actual action is implementing the statement in the `run` method, which is marked with * in Table 4. In order to make the graph clear and simple, we only consider the statement marked with * rather than this event-based calling statement. Figure 29 depicts the PDG of the main procedure.

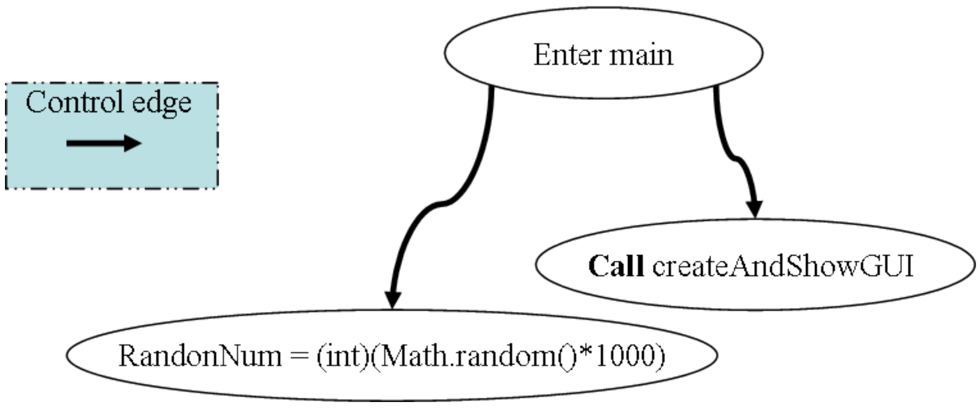


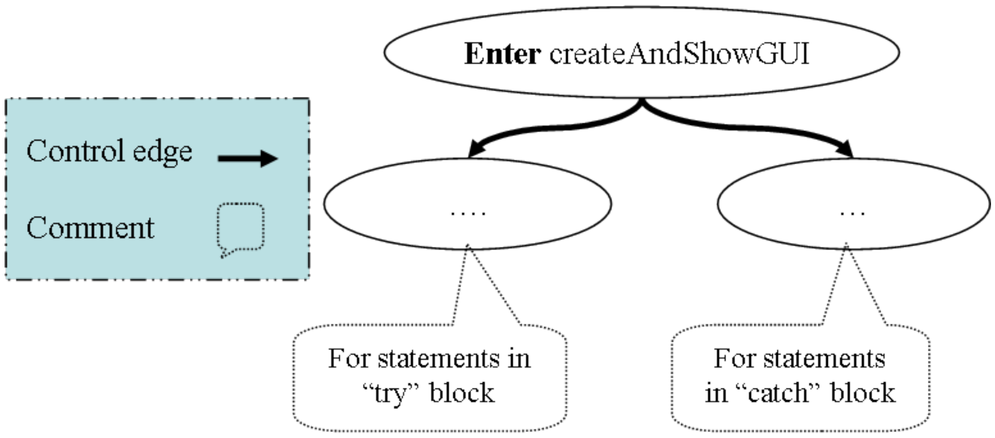
Figure 29: The PDG of main procedure in Segment 6.

The `main` procedure invokes another procedure named `createAndShowGUI`. We build a PDG for this method using the concept of building procedure dependence graphs for any method or inner class discussed above. We show JavaParser's output for `createAndShowGUI` in Table 4 and generate its PDG in Figure 30.

An SDG is a combination of PDGs. To get the SDG of Segment 6, we need to merge the PDG of the `main` procedure and the PDG of the `createAndShowGUI` procedure into the partial SDG in Figure 28. To connect the procedure dependence graphs, we add a call edge between each procedure call vertex and its corresponding procedure entry vertex in its PDG. The connected PDGs are shown in Figure 31.

Figure 31 depicts the parameter-in edge and parameter-out edge between the call vertex and the enter vertex. This is because the procedure `createAndShowGUI` (its source code is given in Appendix A) has several global variables e.g. `number`, `msg` and so on. That means under the call vertex, there are extra vertices containing actual-in and actual-out vertices for each global variable used in this procedure. Likewise, there are extra vertices containing formal-in and formal-out vertices for each global variable under the enter vertex. Thus, parameter-in and parameter-out edges for each global variable exist. Furthermore, the summary edges from each actual-in vertex to its corresponding actual-out vertex exist as well. To keep the graph here simple for reading, we ignore actual-in, actual-out, formal-in, and formal-out vertices in Figure 31 but state the parameter-in and parameter-out edges to indicate that those ignored vertices do exist.

The connected PDGs shown in Figure 31 depict the system dependence graph (SDG) of segment 6 with the system's control dependencies, but without its data dependencies so far. In principle, the SDG of a system is combining the system's data dependence graph and control dependence graph. The data dependencies can be obtained by forwardly traversing the SDG we have built so far.



Enter createAndShowGUI
Control edge
Comment
....
...
For statements in "try" block
For statements in "catch" block

Figure 30: The PDG of the procedure `createAndShowGUI` in Segment 6.

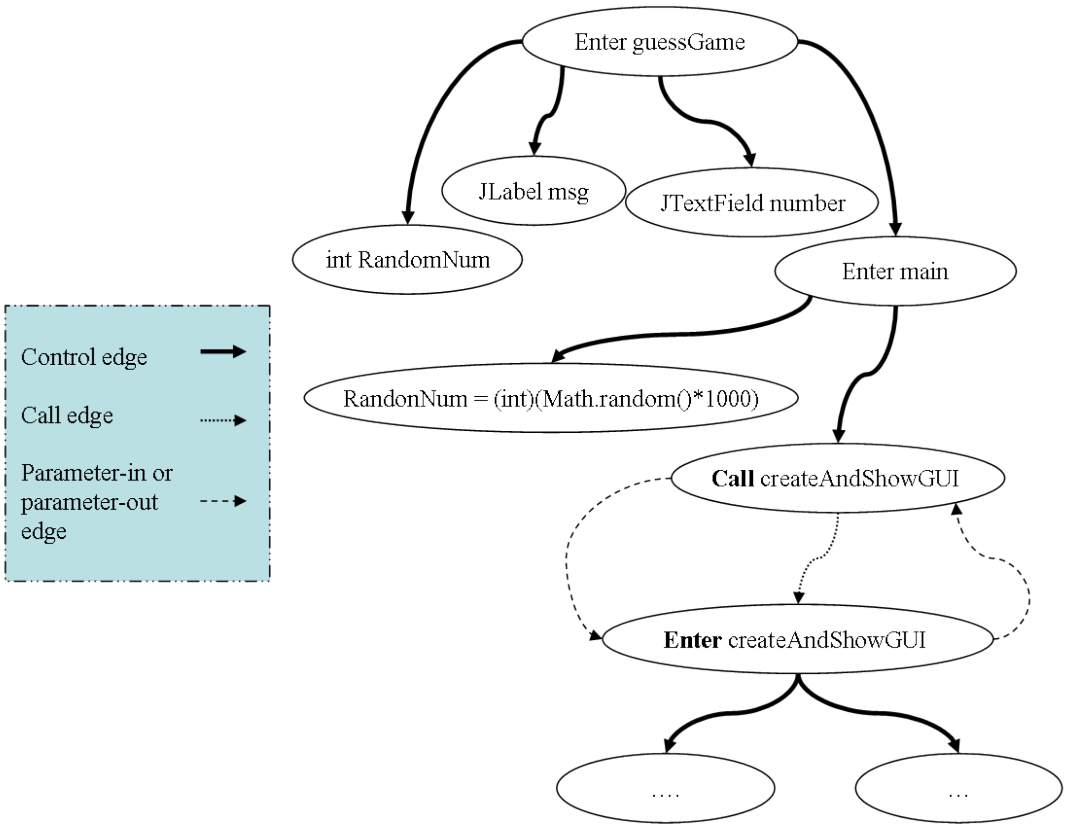


Figure 31: The SDG of Segment 6, connecting PDGs by adding a call edge, parameter-in and parameter-out edges.

The half complete SDG with control flow information contains all of the statements of the source code. In order to build the data dependencies between procedures, we first get a set of object declarations from the AST produced by JavaParser, and traverse the SDG for each declared object from top to bottom and left to right to check if any two vertices have the same declared object. If these exist, we create a data dependence from the top vertex pointing to the lower vertex, or from the left vertex pointing to the right vertex. Besides the interprocedural data dependencies used for the SDG, we also need to complete the PDGs with data dependencies (intraprocedural data dependencies). By a similar method, we first extract each set of variables for each procedure, and then traverse each procedure for each variable of its corresponding set of variables from top to bottom, left to right. If any two vertices have the same variables, we create a data dependence from the top vertex pointing to the lower vertex, or from the left vertex pointing to the right vertex. Thus, we complete the data dependencies of the SDG.

To complete the SDG of Segment 6 in Figure 31, according to the object declarations extracted by JavaParser in Table 4, we start to forwardly traverse the SDG in Figure 31 from the `Enter guessGame` vertex down to the first object `RandomNum`. Once the traversal reaches a node about this object, then we mark this node as *node1*, e.g. the node of `int RandomNum` is *node1*. Then we continue to traverse to the right of *node1*. If there is no related node, we traverse down to the next level of control-flow from left to right. Once we visit a node related to this object, we mark this node as *node2*, e.g. the node of `RandomNum = (int)(…)` is *node2*. Once *node2* has appeared, we add an arrow from node 1 to *node2*, and then unmark *node1* and mark *node2* to be *node1* to continue until we have visited the last node of the SDG. We repeat the traversal for the second object `number`, as well as the third object `msg`, to create their data dependencies. Since there are no variables for the procedures listed in Segment 6, we omit the intraprocedure (inside a procedure) traversal for those variables to create the intraprocedure (between procedures) data dependence, which uses a similar method of creating data dependence of these objects in Table 4. Adding data dependence to the graph of Figure 31 leads to the graph shown in Figure 32.

In brief, we use JavaParser to analyze the control dependencies for the system, build procedure dependence graphs, and then connect those procedure dependence graphs to form the SDG of the system with control dependencies. After that, under the assistance of JavaParser to get a set of object declarations and each set of variables for each procedure, we repeatedly traverse forward in the built SDG for the set of objects or in its PDG for its set of variables to build data dependencies. By now, the complete system dependence graph is built.

As well as the SDG of an application, we also need to define a set of vertices of interest as slicing criteria before we apply the slicing algorithm on the application. A set of vertices of interest for the application can be constructed by the output statements of the application. An output statement refers to a statement in a procedure whose object's value is changed after the statement is executed. For example, in Segment 1, the statement `count = 0` in line 4 is an output statement as the value of the `count` object is changed after the procedure executes this statement. Similarly, the statements from line 4 to line 7 in Segment 1 are output statements. In the SDG, they all can be viewed as vertices of interest if we want to get a program slice with respect to each of them. Basically, we are interested in the outputs and their effects on the state space. Any statement which affects an observation's output is firstly considered as a vertex of interest in the SDG. For instance, the statement for the `count` object discussed above is a kind of statement which affects the output of the current guessing game observation.

With the complete SDG, we apply the two-pass traversal methods to slice the system with a set of vertices of interest of the SDG. What this means is that it does the slicing for the system with a vertex of interest from the set and iterates this process until the set of vertices of interest becomes empty. The program slices generated with respect to the vertices of interest from the set compose the sub-system and then we can gather information from the sub-system to generate our models for the system under analysis.

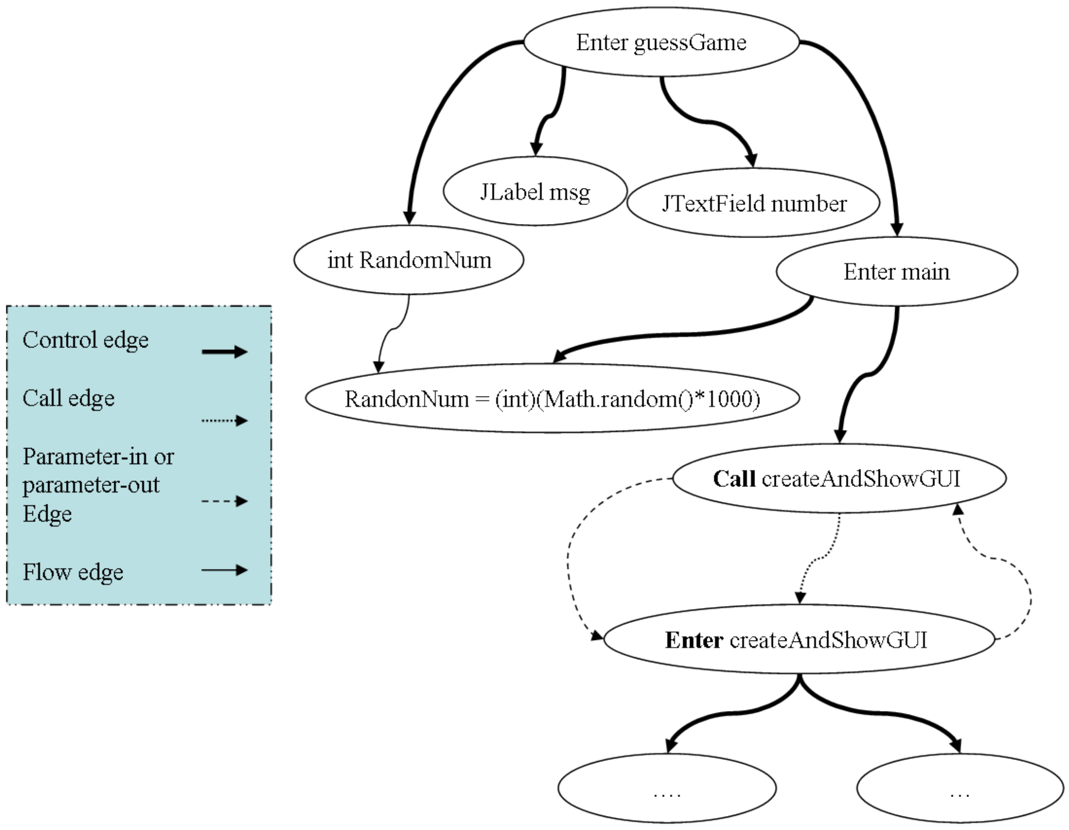


Figure 32: The complete SDG of Segment, creating flow edge represented by a thin arrow to the SDG in Figure 31.

In the next chapter, we describe the results of applying these techniques to the Guess Game example and outline some issues with these techniques.

Chapter 6

Analysis of Slicing & Parsing

We have discussed previously slicing and parsing techniques, and we now discuss how we use the information gathered by those techniques to build our particular models: a presentation model (PModel) and a presentation interaction model (PIM).

A presentation model describes the widgets and their behaviours in a window. Firstly, we need to define how many windows of an application there are. For a Java application, a window can be represented by a frame, a panel or a dialogue. Normally, a frame has one or more panels and the widgets of a window are on panels or directly on the frame. Using parsing techniques, we can extract all of the objects of the application. Appendix C gives the output of the Guess Game from the JavaParser tool. We list the output of the objects of the AST in Table 5.

The first row of Table 5 gives the panels and frames of the example game, and in the second row it gives the relations between the panels and frame. From those, we can define the windows. For example, see the bold phrases in Table 5, the second row shows that `rangePanel` is on `panel` and `panel` is on `frame` (`btnPanel` is on `msgPanel` and `msgPanel` is on `msgFrame`), which means `frame` is a window. From the third row of Table 5, we find what objects are on the window. There are `number` and `go` on `panel`, which means both objects (widgets) are on `frame` (window). Hence, we can extract widgets and their windows by using parsing techniques and then use this information to generate Pmodels.

As well as using the parsing technique for the generation of PModels, we can also traverse the SDG of the slices. First, we use the parsing technique to obtain a set of objects that for panels, frames, or dialogues labelled as *setA* and a set of objects for the others which refers to the widgets on the user interface of the application labelled as *setB*. Assuming we have *setA* and *setB* as below:

setA: panel, frame, rangePanel

setB: number, go

In the sliced SDG, we mark each node (vertex) which declares each item of *setA*, traverse from these marked nodes along their outgoing edges (representing data dependence). If two edges come into the same node, we check the node to see which object is added onto the other one. In Java applications using the Swing library, there is an `add` method for putting a widget on a window. For example, `frame.getContentPane().add(panel)` means `panel` is added onto `frame`; `panel.add(rangePanel)` means `rangePanel` is added onto `panel`. In this way, we can define the `frame` object as a window of the application. We show the inferred structure of this window in Figure 33.

Windows (panels or frames):	JFrame msgFrame JPanel btnPanel **JFrame frame** **JPanel panel** JPanel msgPanel **JPanel rangePanel**
Relations between panels and frames:	**panel On frame.getContentPane()** msgPanel On msgFrame.getContentPane() btnPanel On msgPanel **rangePanel On panel**
Widgets:	number On panel go On panel …

Table 5: A partial parsing output for some objects of Guess Game.

In a similar way, we define the `msgFrame` object as the other window of the application. Hence, we should build two PModels respectively for `frame` and `msgFrame`. After defining windows, we need to extract the information about the widgets of each window. In a similar manner to defining windows above, we mark those nodes which represent the declarations of each item in *setB* and traverse from these marked nodes along the outgoing edges. If a node visited has more than one incoming edge we mark this node as *nodeX*, we check *nodeX* to see if it has an `add` method and find out which object calls the method. For example, if *nodeX* represents the statement `panel.add(go)`, the object `panel` calls the `add` method to add the object `go`. That means `go` is on `panel` and `panel` is on `frame`, so the widget `go` is on `frame`. Alternatively, once we get *nodeX*, we traverse backward from this node to see if any of the marked nodes from *setA* can be reached. If a marked node from *setA* is reached, it means *nodeX* is on the item the marked node refers to, and then use the same way as we defined windows to decide which window the *nodeX* is on. Based on Table 5, the inferred structure of `frame` is shown in Figure 34.

A PModel also defines the behaviour of each widget of the window. For a Java application using the Swing library, event-based control implies the widget is an ActionControl, i.e. it generates some behaviours. If we refer to Segment 2 showing the event-based control for `go`, it indicates the widget `go` is an ActionControl and has a behaviour defined by calling the `CheckListener` procedure. Using the widget `go` as an example, we traverse forward from the node which declares this widget and check each visited node, to identify if any one represents an event-based control. Once this kind of node is visited, we traverse its procedure call to identify if the behaviour is an S_behaviour, or I_behaviour, or S_behaviour and I_behaviour. I_behaviours navigate the windows of the application. For our example application, the behaviour for opening the message window in the form of `msgFrame` controlled by the statement `msgFrame.setVisible(true)` *statA*, or hiding the message window controlled by the statement `msgFrame.setVisible(false)` *statB*, are I_behaviours. Traversing the procedure calls, if it only has a node about *statA* or *statB*, then this behaviour is an I_behaviour; if it also has other nodes, then this has multiple behaviours and is both of S_behaviour and I_behaviour; if it has nodes but nothing about *statA* or *statB*, then this behaviour is an S_behaviour. Using this algorithm, we can define the behaviours for the widgets which have event-based controls. The widget `go` has both an I_behaviour and S_behaviour, as its called procedure `CheckListener` contains *statA* (the procedure's source code is given in Appendix A), so in the SDG of slices we must have a node representing it in the procedure call. We can name the I_behaviour as I_openMsgWin, and the S_behaviour as S_checkNumbers. For the other widgets, we categorise them by their type, such as "Entry" for widgets which are textfields, "StatusDisplay" for widgets which are labels etc. Using JavaParser, we can get object declarations for global variables or variable declarations for variables. In Table 4 in Section 5.3, there is a row for object declarations. This gives an output which is `JTextField number`, which means the widget `number` is a textfield, so we can categorise it as an "Entry".

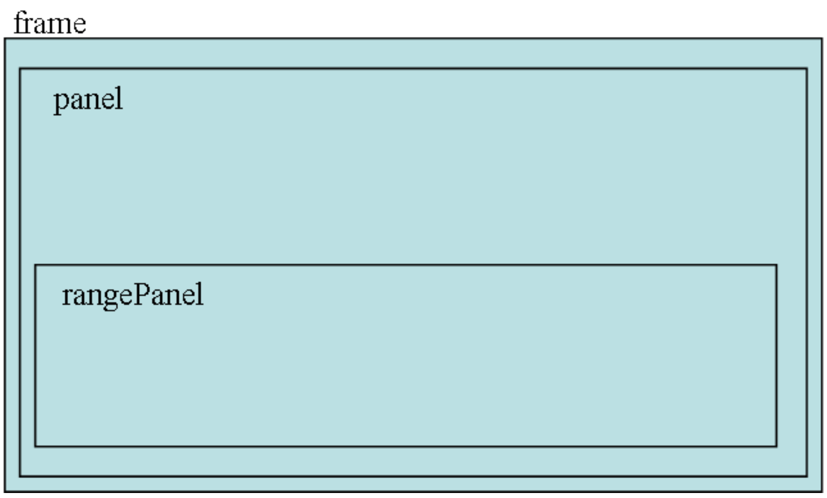


Figure 33: The inferred structure of a window in the form of a `frame` in the source code without widgets.

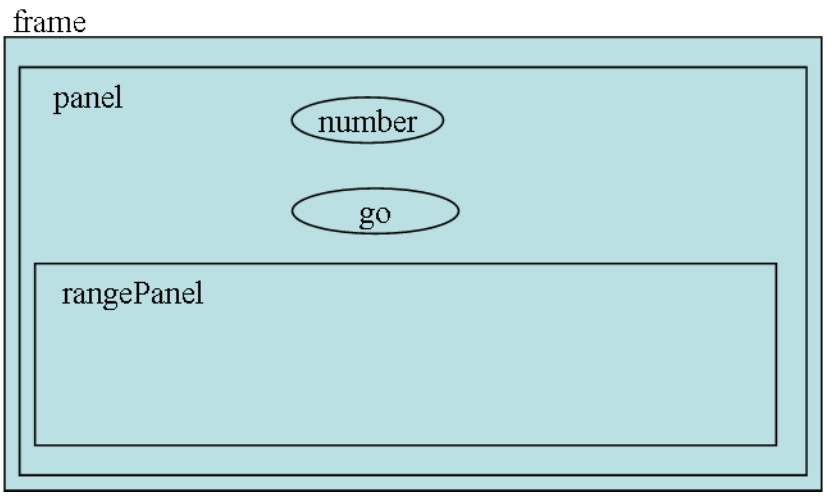


Figure 34: The inferred structure of a window in the form of frame in the source code with widgets, based on Table 5.

Now, we have clarified windows, widgets, and their behaviours for Table 5. Based on the results we get by the methods above, we can build the PModels shown in

Table 6.

frame is	(number, Entry, ()) (go, ActionControl, (I_openMsgFrame, S_checkNumbers)) ….
msgFrame is	….

Table 6: The PModel of frame in Table 5.

We can check the source code in Appendix A to know that `frame` represents the main window and `msgFrame` represents the message window of the application. According to Table 6, we also can build the PIM shown in Figure 35.

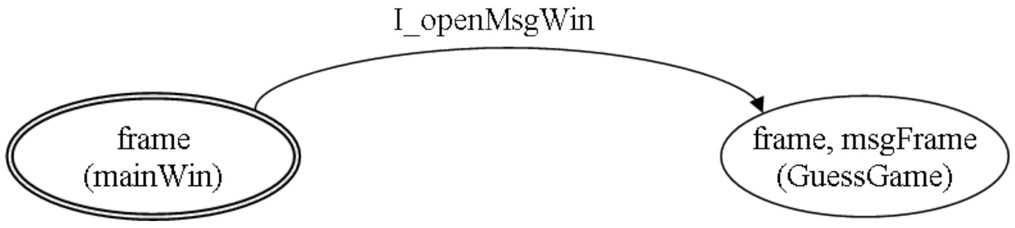


Figure 35: The PIM based on Table 6.

Using the slicing and parsing techniques with the SDG of the slices, we are able to extract the information for the generation of presentation models and a presentation interaction model using the approach we have discussed above. We apply the approach based on the SDG of the slices of an application under analysis (we could apply the approach on the whole SDG of an application under analysis but it is too time-consuming to traverse the complete SDG as it is large and complex). What this means is that if the SDG of the slices is not precise it can lead to mistakes when we build the models. Also, in the process of slicing and parsing for getting slices of the application, some issues may occur, including those caused by the style of coding.

Using the techniques described, we may get sub-programs which are too big and contain too much information. For example, the information about the event of a button may consist of hundreds of lines of code including several functions in one method, or some useless information about extraneous widgets which were used for testing during the programming period and which do not work in the final application (i.e. the widgets that are never added to any of the windows). In this case, we will build a big procedure dependence graph, and then get a big system dependence graph which is a collection of procedure dependence graphs. Big system dependence graphs will cause difficulties in traversal for creating relationships between the vertices and also take a long time to do. Similarly, it is difficult and time-consuming when we use the two-passes method (details in Section 5.2) on a large graph, traversing pass 1 and pass 2 to get the sliced program.

The second problem is that we may extract too little information leading to models that are not understandable, such as getting some event name only which by itself is meaningless. For example, if we end up with a behaviour in the model called "methodA", this gives no clue as to what the method does and will make it harder to understand the method. To make sense of this, we will need to look at the code of the method to confirm behaviour (i.e. we end up having to manually read the source code again). This causes trouble in reading and understanding the models, consumes extra time, and also requires human involvement.

Another problem is about hidden information relating to the system's response, which may be missed because we only focus on extracting information about the graphical user interface only. The model of an interactive system is not only the model of the graphical user interface but also the model of the system's functionality. Thus, we also need to get the sliced program where slices relate to the hidden information (for example, global variables). This problem relates to the set of properties of interest which may not be complete as slicing criteria.

The first two problems are caused by the application programmers' coding style, and the last problem is the result of the process of slicing and parsing because the analysts only focus their attention on the graphical user interface of an interactive system leading to the incomplete slicing criteria.

Coding style is critical. If we are to build tools which fully automate the process (which is desirable) then in order to capture all information, we need to consider all of the possible ways the program may be designed. This is not feasible, so coding style is hard to address and the kind of issues caused by different coding styles may occur and lead us to get imprecise models of the application. However, the second problem relating to slicing criteria can be solved by a more structured slicing criteria generation. There may still be underlying functionality we do not identify, but if this is only the behaviour not required for our models (because the user can not access it via interaction) then that does not cause problems for the PModels or PIM.

The presentation model relation (PMR) of the application can not be built following the methods described, as it is a relation between the presentation models and a specification of the application, but we do not have access to the specification during slicing or parsing.

Chapter 7

Conclusions

7.1 Overview of Project Goals

This project discusses reverse engineering interactive software applications, which refers to applications with user interfaces, by static methods. Using existing reverse engineering techniques to get exact models of interactive software systems is not easy. Many existing reverse engineering techniques only focus on the system functionality or the system's user interface, and generate very big models which cause slow analysis and which are resource intensive. This project addresses those problems. We can reverse engineer the interactive system to get both the structure of the graphical user interface and its corresponding internal behaviours, and have identified how slicing and parsing techniques could be used to generate a particular set of models, presentation models and presentation interaction models [4].

7.2 Summary of Results

With the aim of building small graphs of a system to derive the models, this project suggests using program slicing techniques combined with parsing techniques to get the sub-system which contains all of the necessary information about the system's user interface but without additional irrelevant information for the models' generation. We first build the system dependence graph of the system which describes the data dependence graph and control dependence graph of the system. Next we define a set of vertices of interest as slicing criteria which normally is a set of the outputs of the system, and then we use the two-passes method to slice with respect to each vertex of interest to get the corresponding slice. Finally, we combine these slices to form the program slice of the system with the slicing criteria of the set of vertices of interest.

Based on the program slice, we suggest an approach of using parsing and slicing techniques to extract the user interface's structure from the program slice for the purpose of building the particular set of models we are interested in. The models are presentation models that formally describe the behaviours and components of the user interface of an interactive system and a presentation interaction model that gives the availability and navigation of the system's user interfaces to ensure each user interface of the system is reachable by the user. The set of models represent an abstraction of an interactive system, including both the user interface and system functionality.

We have examined these techniques and approaches with our study case – Guess Game throughout this project.

There are some issues which can occur during the generation of program slices because of the system programmers' code style and the choice of the slicing criteria. Hence, the program slices may vary and lead to differences in the final models generated. However, we have shown that such an approach is feasible and that our idea of combining parsing and slicing in the manner shown is a valuable contribution and presents a solution to the problem.

7.3 Future Work

This project uses slicing and parsing to aid reverse engineering techniques for interactive systems. We have used parsing tools and then manually generated the system dependence graph. In future, we consider developing a tool (based on the algorithms described in Chapter 5) to automatically create the system dependence graph for an interactive system to be analysed. In addition, we consider that it may be beneficial to combine some dynamic analysis methods to improve our techniques. For example, the technique used by Systa [27] uses a debugger to get event traces and outputs state diagrams about the events which can be used to examine the total behaviour of a class, object, or method. We consider this may be helpful for improving the generation of the system dependence graphs.

8 Bibliography

[1]. Leveson, N. G. (1995). *SafeWare: System safety and computers*. Reading, Mass.: Addison-Wesley.

[2]. *Source making: Visitor Design Pattern*. (2012). Retrieved from http://sourcemaking.com/design_patterns/visitor

[3]. Chikofsky, E. J., & Cross, J. H. (1990). Reverse engineering and design recovery - A taxonomy. *IEEE Software, 7*(1), 13-17.

[4]. Bowen, J., & Reeves, S. (2008). Formal models for user interface design artefacts. *Innovations in Systems and Software Engineering, 4*(2), 125-141.

[5]. *Wikipedia: User-centered design*. (2012). Retrieved from http://en.wikipedia.org/wiki/User-centered_design

[6]. Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C. S., Robby, & Hongjun, Z. (2000). Bandera: Extracting finite-state models from Java source code. *Proceedings of International Conference on Software Engineering, June 4 - 11, 2000* (pp. 439-448). New York: ACM.

[7]. Memon, A., Banerjee, I., & Nagarajan, A. (2003). GUI ripping: Reverse engineering of graphical user interfaces for testing. *Proceedings - 10th Working Conference on Reverse Engineering, November 13 - 16, 2003* (pp. 260-269). Victoria, BC, Canada: IEEE Computer Society.

[8]. Samir, H., Stroulia, E., & Kamel, A. (2007). Swing2Script: Migration of Java-Swing applications to Ajax Web applications. *Proceedings - 14th Working Conference on Reverse Engineering, WCRE 2007, October 28 - 31, 2007* (pp. 179-188). Vancouver, BC, Canada: IEEE Computer Society.

[9]. Gimblett, A., & Thimbleby, H. (2010). User interface model discovery: Towards a generic approach. *Proceedings - 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS'10, June 19 - 23, 2010 (pp. 145-154)*. Berlin, Germany: ACM.

[10]. Paiva, A. C. R., Faria, J. C. P., Tillmann, N., & Vidal, R. A. M. (2005). A model-to-implementation mapping tool for automated model-based GUI testing. *Proceedings - Formal Methods and Software Engineering. 7th International Conference on Formal Engineering Methods, ICFEM 2005. November 1 - 4, 2005* (pp. 450-464). Berlin, Germany: Springer-Verlag.

[11]. Bandelloni, R., Patern, F., & Santoro, C. (2008). Reverse engineering cross-modal user interfaces for ubiquitous environments. In G. Jan, H. M. Borup, P. Philippe, C. V. Gerrit & W. Janet (Eds.), *Engineering Interactive Systems* (pp. 285-302). Berlin: Springer.

[12]. Dwyer, M. B., Robby, Tkachuk, O., & Visser, W. (2004). Analyzing interaction orderings with model checking. *Proceedings - 19th International Conference on Automated Software Engineering, ASE 2004, September 20 - 24, 2004* (pp. 154-163). Linz, Austria: IEEE Computer Society.

[13]. Paiva, A. C. R., Faria, J. C. P., & Mendes, P. M. C. (2008). Reverse engineered formal models for GUI testing. *Proceedings - Formal Methods for Industrial Critical Systems. 12th International Workshop, FMICS 2007, July 1 - 2, 2007* (pp. 218-233). Berlin, Germany: Springer-Verlag.

[14]. Silva, J. C., Campos, J. C., & Saraiva, J. (2006). Combining formal methods and functional strategies regarding the reverse engineering of interactive applications. *Proceedings - Interactive Systems: Design, Specification, and Verification. 13th International Workshop, DSVIS 2006. July 26 - 28, 2006* (pp. 137-150). Berlin, Germany: Springer-Verlag.

[15]. Silva, J. C., Saraiva, J., & Campos, J. C. (2009). A generic library for GUI reasoning and testing. *Proceedings - 24th Annual ACM Symposium on Applied Computing, SAC 2009, March 8 - 12, 2009* (pp. 121-128). Honolulu: ACM.

[16]. Staiger, S. (2007). Reverse engineering of graphical user interfaces using static analyses. *Proceedings - 14th Working Conference on Reverse Engineering, October 28 - 31, 2007* (pp. 189-198). Piscataway, NJ: IEEE.

[17]. Silva, J. C., Silva, C., Goncalo, R., Saraiva, J., & Campos, J. C. (2010). The GUISurfer tool: Towards a language independent approach to reverse engineering GUI code. *Proceedings - 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS'10, June 19 - 23, 2010* (pp. 181-186). Berlin, Germany: ACM.

[18]. Silva, J. C., Saraiva, J., & Campos, J. C. (2011). *Models for the reverse engineering of Jave/Swing application*. Retrieved from http://news.informatik.uni-mainz.de/ALT/Dateien/26-Silva-old.pdf

[19]. Dwyer, M. B., Carr, V., & Hines, L. (1997). Model checking graphical user interfaces using abstractions. *Proceedings - 6th European Software Engineering Conference, ESEC/FSE '97, September 22 - 25, 1997* (6 ed., pp. 244-261). USA: ACM.

[20]. Memon, A. M., Soffa, M. L., & Pollack, M. E. (2001). Coverage criteria for GUI testing. *Proceedings - 8th European Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9), September 10 - 14, 2001* (pp. 256-267). Vienna, Austria: ACM.

[21]. Moore, M. M. (1996). Rule-based detection for reverse engineering user interfaces. *Proceedings of WCRE '96: 3rd Working Conference on Reverse Engineering, November 8 - 10, 1996* (pp. 42-48). Los Alamitos, CA: IEEE Computer Society.

[22]. Horwitz, S., Reps, T., & Binkley, D. (2004). Interprocedural slicing using dependence graphs. *ACM SIGPLAN Notices, 39*(4), 229-243.

[23]. Agrawal, H. (1994). On slicing programs with jump statements. *ACM SIGPLAN Notices, 29*(6), 302-312.

[24]. Reps, T., Horwitz, S., Sagiv, M., & Rosay, G. (1994). Speeding up slicing. *Proceedings – 2nd ACM SIGSOFT Symposium on Foundations of Software Enginering. SIGSOFT '94, December 6 - 9, 1994* (pp. 11-20). USA: ACM.

[25]. Tip, F. (1994). *A survey of program slicing techniques*. Retrieved from http://researcher.ibm.com/files/us-ftip/jpl1995.pdf

[26]. Weiser, M. (1981). *Program slicing.* Paper presented at the Proceedings - 5th International Conference on Software Engineering, March 9 - 12, 1981, New York.

[27]. Systa, T. (1999). *Dynamic reverse engineering of Java software.* Retrieved from http://www.cs.tut.fi/~tsysta/papers/ecoopnew.pdf

[28]. Scheetz, M., von Mayrhauser, A., & France, R. (1999). Generating test cases from an OO model with an AI planning system. *Proceedings - 10th International Symposium on Software Reliability Engineering, November 1 - 4, 1999* (pp. 250-259). Los Alamitos, CA: IEEE Computer Society.

[29]. Fu, C., Grechanik, M., & Xie, Q. (2009). Inferring types of references to GUI objects in test scripts. *Proceedings - 2nd International Conference on Software Testing, Verification, and Validation, ICST 2009, April 1 - 4, 2009* (pp. 1-10). Denver, CO: IEEE Computer Society.

[30]. Grilo, A. M. P., Paiva, A. C. R., & Faria, J. P. (2006). *Reverse engineering of GUI models*. Retrieved from http://inforum.org.pt/INForum2009/docs/full/paper_39.pdf

[31]. Lutteroth, C. (2008). Automated reverse engineering of hard-coded GUI layouts. *Proceedings of the 9th conference on Australasian user interface, AUIC2008, Wollongong, Australia* (pp. 65-73). Darlinghurst, Australia: Australian Computer Society.

[32]. Systa, T. (1999). On the relationships between static and dynamic models in reverse engineering Java software. *Proceedings of the 6th Working Conference on Reverse Engineering, Atlanta, Georgia* (pp. 304-313): IEEE.

[33]. Xun, Y., Cohen, M. B., & Memon, A. M. (2011). GUI Interaction testing: Incorporating event context. *IEEE Transactions on Software Engineering, 37*(4), 559-574.

[34]. Moore, M. (1995). *Reverse engineering user interfaces: A technique*. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.49.5043&rep=rep1&type=pdf

[35]. Da Cruz, A. M. R., & Faria, J. P. (2010). *Automatic generation of user interface models and prototypes from domain and use case models*. Retrieved from http://www.mendeley.com/research/automatic-generation-user-interface-models-prototypes-domain-case-models/

[36]. ISO/IEC 13568. (2002). *Information technology -- Z formal specification notation -- Syntax, type system and semantics*.

[37]. Reeve, G. (2005). *A refinement theory for μCharts*. PhD thesis, University of Waikato, Hamilton, New Zealand.

[38]. Bowen, J., & Reeves, S. (2009). Refinement for user interface designs. *Formal Aspects of Computing, 21*(6), 589-612.

9 Appendix A

The Source Code of the Guess Game in Java is shown here.

```
/**
author: Feifei Lin
*/

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class guessGame {
    private static int RandomNum;
    private JTextField number;
    private JLabel msg;

    private JFrame msgFrame;
    private JPanel btnPanel;
    private int count = 0; // how many times to get the right number
    private JButton continueBtn; // click to back the main window or reset the game

    private JRadioButton range1;
    private JRadioButton range2;
    private JRadioButton range3;

    private JLabel countMsg;

    public static void main(String[] args) {
        RandomNum = (int)(Math.random() *1000);

        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new guessGame().createAndShowGUI();
            }
            });
        }

        public void createAndShowGUI() {
            try {
                JFrame frame = new JFrame("Guess A Number Game");
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

                JPanel panel = new JPanel();
                panel.setPreferredSize(new Dimension(450,150));
                frame.getContentPane().add(panel);

                JLabel label1 = new JLabel("I am thinking of a number X where: " + "0 <= X < 1000, Guess what I am: ");
                number = new JTextField("Guess a number", 20);
                JButton go = new JButton("Go");
                msg = new JLabel();

                panel.add(label1);
                panel.add(number);
                panel.add(go);
                go.addActionListener(new CheckListener());

                msgFrame = new JFrame("The Message of The Number");
                msgFrame.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);
                JPanel msgPanel = new JPanel();
                msgPanel.setPreferredSize(new Dimension(400,150));
                msgFrame.getContentPane().add(msgPanel);
                msgPanel.setLayout(new BoxLayout(msgPanel, BoxLayout.PAGE_AXIS));
                msgPanel.add(msg);
                msgPanel.add(Box.createRigidArea(new Dimension(0,5)));

                msgPanel.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));

                btnPanel = new JPanel();
                btnPanel.setLayout(new BoxLayout(btnPanel, BoxLayout.LINE_AXIS));

                btnPanel.setBorder(BorderFactory.createEmptyBorder(15,10,10,10));
                btnPanel.setAlignmentX(Box.LEFT_ALIGNMENT);
                JButton exitBtn = new JButton(″Exit Game″);
                exitBtn.addActionListener(new FinishGame());
                continueBtn = new JButton(″Continue Game″);
                continueBtn.addActionListener(new ContinueGame());

                btnPanel.add(exitBtn);
                btnPanel.add(Box.createRigidArea(new Dimension(10,0)));
                btnPanel.add(continueBtn);

                msgPanel.add(btnPanel, BorderLayout.CENTER);
                msgPanel.add(Box.createRigidArea(new Dimension(0,5)));
                countMsg = new JLabel();
                msgPanel.add(countMsg, BorderLayout.PAGE_END);

                msgFrame.pack();
                msgFrame.setVisible(false);

                JMenuBar menuBar;
                JMenu menu;
                JMenuItem menuItem;

                menuBar = new JMenuBar();
                menu = new JMenu(″File″);

                menu.setMnemonic(KeyEvent.VK_A);

                menu.getAccessibleContext().setAccessibleDescription(″The only menu in this program that has menu items″);
                menuBar.add(menu);

                menuItem = new JMenuItem(″New Round″, KeyEvent.VK_N);
                menuItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_2, ActionEvent.ALT_MASK));
                menuItem.addActionListener(new NewRound());
                menu.add(menuItem);

                menuItem = new JMenuItem(″Close″, KeyEvent.VK_Q);
                menuItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_1, ActionEvent.ALT_MASK));
```

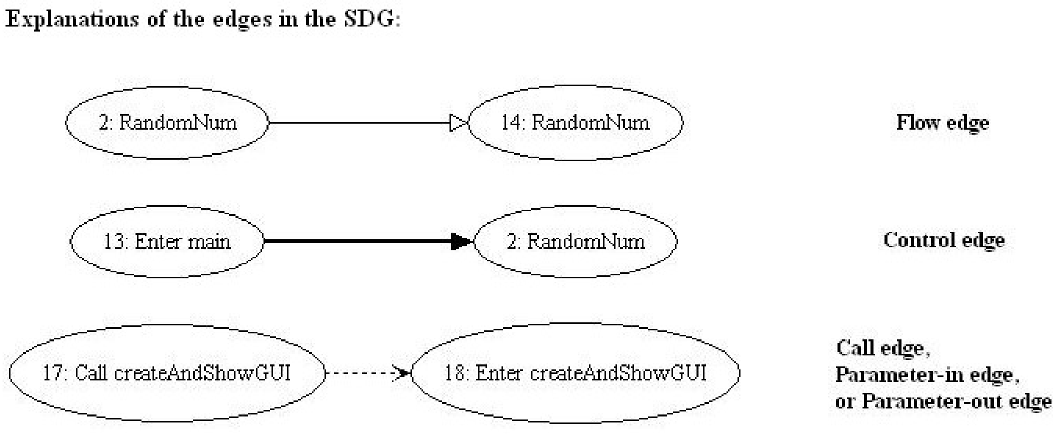
```
            menuItem.addActionListener(new FinishGame());
            menu.add(menuItem);

            frame.setJMenuBar(menuBar);

            // range:
            // radio buttons
            range1 = new JRadioButton("0 <= X < 100");
            range2 = new JRadioButton("0 <= X < 5000");
            range3 = new JRadioButton("RESET");
            range3.setVisible(false);
            ButtonGroup group = new ButtonGroup();
            group.add(range1);
            group.add(range2);
            group.add(range3);
            range1.addActionListener(new NewRound());
            range2.addActionListener(new NewRound());
            range3.addActionListener(new NewRound());

            JPanel rangePanel = new JPanel();
            rangePanel.setPreferredSize(new Dimension(400,70));
           rangePanel.setBorder(BorderFactory.createTitledBorder
  ("Range Options"));

            panel.add(rangePanel);
            rangePanel.add(range1);
            rangePanel.add(range2);
            rangePanel.add(range3);

            frame.pack();
            frame.setVisible(true);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void generateRandomNumber() {
        if (range1.isSelected()) {
            RandomNum = (int)(Math.random() *100);
            range3.setVisible(true);
        }
    else if (range2.isSelected()) {
        RandomNum = (int)(Math.random() *5000);
        range3.setVisible(true);
    }
    else {
        RandomNum = (int)(Math.random() *1000);
        range3.setVisible(false);
    }
}

public void NewGame() {
    generateRandomNumber();
    number.setText("Guess a number");
    count = 0;
    continueBtn.setText("Continue Game");
    msgFrame.setVisible(false);
    countMsg.setVisible(false);
}

public class NewRound implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        NewGame();
    }
}

public class ContinueGame implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        // TODO Auto-generated method stub
        if (continueBtn.getText() == "Next Round") {
            NewGame();
        }
        msgFrame.setVisible(false);
    }
}

public class FinishGame implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        // TODO Auto-generated method stub
        System.exit(0);
    }
    }

    public class CheckListener implements ActionListener {
        public void actionPerformed(ActionEvent ev){
            msgFrame.setVisible(true);
            count ++;
            System.out.println("You have guessed "+ count + " times");

            String str = number.getText();
            try {
                int guessNum = Integer.parseInt(str);


                if (guessNum < RandomNum) {
                    msg.setText("It's LOWER than what I think, please guess again!");
                } else if (guessNum > RandomNum) {
                    msg.setText("It's HIGHER than what I think, please guess again!");
                } else {
                    msg.setText("Congratulations! The number I'm thinking is " + RandomNum + ".");
                    continueBtn.setText("Next Round");
                    countMsg.setVisible(true);
                    countMsg.setText("You guessed " + count + " times for this round.");
                }
            } catch (NumberFormatException e) {
                return;
            }
        }
    }
}
```

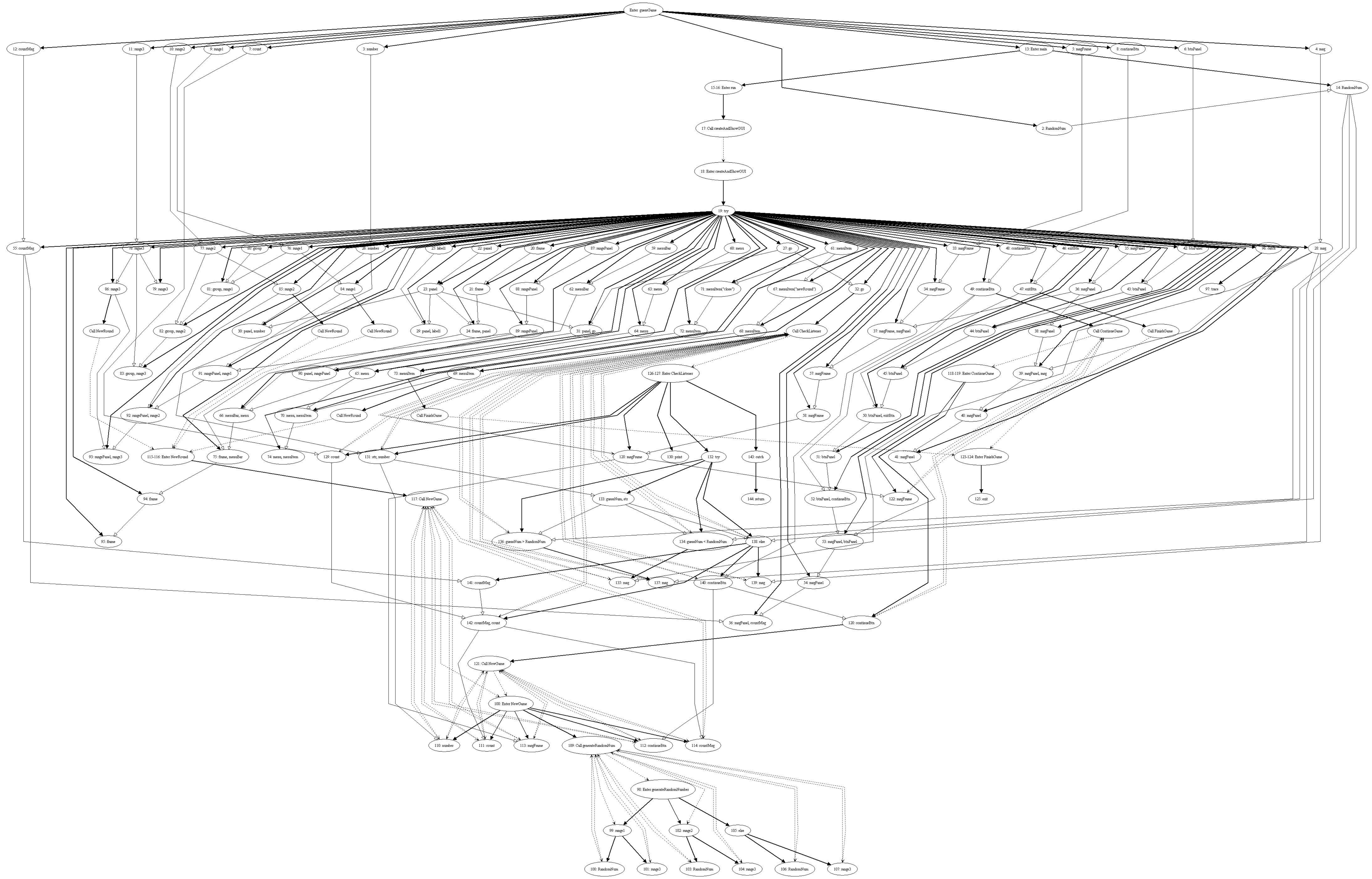
10 Appendix B

The system dependence graph of Guess Game is shown on the following insert.

In order to keep the graph simple for reading, the extra vertices including the actual-in, actual-out vertices for the global variables for call statements are omitted, as well as the extra vertices as the property of the Swing library. However, we keep the relations for the global variables between call statements and called procedures by showing their parameter-in and parameter-out edges.

Based on the source code of Guess Game in Appendix A, we put a number of the line of a statement with the variables used in the statement to express a node in the SDG on the following insert. That means to read the full expression of a node in Appendix A.





Enter: guessGame
12: countMsg
11: range3
10: range2
9: range1
7: count
3: number
13: Enter main
5: msgFrame
8: continueBtn
6: btnPanel
4: msg
15-16: Enter run
14: RandomNum
17: Call createAndShowGUI
2: RandomNum
18: Enter createAndShowGUI
19: try
55: countMsg
78: range3
77: range2
80: group
76: range1
26: number
25: label1
22: panel
20: frame
87: rangePanel
59: menuBar
60: menu
27: go
61: menuItem
33: msgFrame
48: continueBtn
46: exitBtn
35: msgPanel
42: btnPanel
96: catch
28: msg
86: range3
79: range3
81: group, range1
85: range2
84: range1
23: panel
21: frame
88: rangePanel
62: menuBar
63: menu
71: menuItem("close")
67: menuItem("newRound")
32: go
34: msgFrame
49: continueBtn
47: exitBtn
36: msgPanel
43: btnPanel
97: trace
Call NewRound
82: group, range2
30: panel, number
Call NewRound
Call NewRound
29: panel, label1
24: frame, panel
89: rangePanel
31: panel, go
64: menu
72: menuItem
68: menuItem
Call CheckListener
37: msgFrame, msgPanel
44: btnPanel
38: msgPanel
Call ContinueGame
Call FinishGame
83: group, range3
91: rangePanel, range1
90: panel, rangePanel
65: menu
73: menuItem
69: menuItem
126-127: Enter CheckListener
37: msgFrame
45: btnPanel
118-119: Enter ContinueGame
39: msgPanel, msg
92: rangePanel, range2
66: menuBar, menu
70: menu, menuItem
Call NewRound
Call FinishGame
38: msgFrame
50: btnPanel, exitBtn
40: msgPanel
93: rangePanel, range3
115-116: Enter NewRound
75: frame, menuBar
74: menu, menuItem
129: count
131: str, number
128: msgFrame
130: print
132: try
143: catch
51: btnPanel
41: msgPanel
123-124: Enter FinishGame
94: frame
117: Call NewGame
133: guessNum, str
144: return
52: btnPanel, continueBtn
122: msgFrame
125: exit
95: frame
136: guessNum &gt; RandomNum
134: guessNum &lt; RandomNum
138: else
53: msgPanel, btnPanel
141: countMsg
135: msg
137: msg
140: continueBtn
139: msg
54: msgPanel
142: countMsg, count
56: msgPanel, countMsg
120: continueBtn
121: Call NewGame
108: Enter NewGame
110: number
111: count
113: msgFrame
109: Call generateRandomNum
112: continueBtn
114: countMsg
98: Enter generateRandomNumber
99: range1
102: range2
105: else
100: RandomNum
101: range3
103: RandomNum
104: range3
106: RandomNum
107: range3

11 Appendix C

The main outputs of parsing Guess Game by JavaParser are listed below, extracting inner classes, method declarations, object declarations, and variable declarations used in the whole program.

For inner classes:

```
$ inner class: public class NewRound implements ActionListener {

    public void actionPerformed(ActionEvent e) {
        NewGame();
    }
}
$ inner class: public class ContinueGame implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent e) {
        if (continueBtn.getText() == "Next Round") {
            NewGame();
        }
        msgFrame.setVisible(false);
    }
}
$ inner class: public class FinishGame implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
}
$ inner class: public class CheckListener implements ActionListener {

    public void actionPerformed(ActionEvent ev) {
        msgFrame.setVisible(true);
        count++;
        System.out.println("You have guessed " + count + " times");
        String str = number.getText();
        try {
            int guessNum = Integer.parseInt(str);
            if (guessNum < RandomNum) {
                msg.setText("It's LOWER than what I think, please guess
again!");
            } else if (guessNum > RandomNum) {
                msg.setText("It's HIGHER than what I think, please guess
again!");
            } else {
                msg.setText("Congratulation! The number I'm thinking is " +
RandomNum + ".");
                continueBtn.setText("Next Round");
                countMsg.setVisible(true);
                countMsg.setText("You guessed " + count + " times for this
round.");
            }
        } catch (NumberFormatException e) {
            return;
        }
    }
}
```

For global variables:

```
Object Declaration: int RandomNum
FieldDeclaration: private static int RandomNum;
Object Declaration: JTextField number
FieldDeclaration: private JTextField number;
Object Declaration: JLabel msg
FieldDeclaration: private JLabel msg;
Object Declaration: JFrame msgFrame
FieldDeclaration: private JFrame msgFrame;
Declaration of the Window class: msgFrame
Object Declaration: JPanel btnPanel
FieldDeclaration: private JPanel btnPanel;
Declaration of the Window class: btnPanel
Object Declaration: int count
FieldDeclaration: private int count = 0;
Object Declaration: JButton continueBtn
FieldDeclaration: private JButton continueBtn;
Object Declaration: JRadioButton range1
```

```
FieldDeclaration: private JRadioButton range1;
Object Declaration: JRadioButton range2
FieldDeclaration: private JRadioButton range2;
Object Declaration: JRadioButton range3
FieldDeclaration: private JRadioButton range3;
Object Declaration: JLabel countMsg
FieldDeclaration: private JLabel countMsg;
```

For method declarations:

```
# method name: main
! modifiers: 9
! type: void
! parameter: [String[] args]
! body-blockstmt: {
    RandomNum = (int) (Math.random() * 1000);
    SwingUtilities.invokeLater(new Runnable() {

        public void run() {
            new guessGame().createAndShowGUI();
        }
    });
}
# method name: createAndShowGUI
! modifiers: 1
! type: void
! parameter: null
! body-blockstmt: {
    try {
        JFrame frame = new JFrame("Guess A Number Game");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel panel = new JPanel();
        panel.setPreferredSize(new Dimension(450, 150));
        frame.getContentPane().add(panel);
        JLabel label1 = new JLabel("I am thinking of a number X where: " +
"0 <= X < 1000, Guess what I am: ");
        number = new JTextField("Guess a number", 20);
        JButton go = new JButton("Go");
        msg = new JLabel();
        panel.add(label1);
        panel.add(number);
        panel.add(go);
        go.addActionListener(new CheckListener());
        msgFrame = new JFrame("The Message of The Number");
        msgFrame.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);
        JPanel msgPanel = new JPanel();
        msgPanel.setPreferredSize(new Dimension(400, 150));
        msgFrame.getContentPane().add(msgPanel);
        msgPanel.setLayout(new BoxLayout(msgPanel, BoxLayout.PAGE_AXIS));
        msgPanel.add(msg);
        msgPanel.add(Box.createRigidArea(new Dimension(0, 5)));
        msgPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10,
10));
        btnPanel = new JPanel();
        btnPanel.setLayout(new BoxLayout(btnPanel, BoxLayout.LINE_AXIS));
        btnPanel.setBorder(BorderFactory.createEmptyBorder(15, 10, 10,
10));
        btnPanel.setAlignmentX(Box.LEFT_ALIGNMENT);
        JButton exitBtn = new JButton("Exit Game");
        exitBtn.addActionListener(new FinishGame());
        continueBtn = new JButton("Continue Game");
        continueBtn.addActionListener(new ContinueGame());
        btnPanel.add(exitBtn);
        btnPanel.add(Box.createRigidArea(new Dimension(10, 0)));
        btnPanel.add(continueBtn);
        msgPanel.add(btnPanel, BorderLayout.CENTER);
        msgPanel.add(Box.createRigidArea(new Dimension(0, 5)));
        countMsg = new JLabel();
        msgPanel.add(countMsg, BorderLayout.PAGE_END);
        msgFrame.pack();
        msgFrame.setVisible(false);
        JMenuBar menuBar;
        JMenu menu;
        JMenuItem menuItem;
        menuBar = new JMenuBar();
        menu = new JMenu("File");
        menu.setMnemonic(KeyEvent.VK_A);
        menu.getAccessibleContext().setAccessibleDescription("The only menu
in this program that has menu items");
        menuBar.add(menu);
        menuItem = new JMenuItem("New Round", KeyEvent.VK_N);
        menuItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_2,
ActionEvent.ALT_MASK));
        menuItem.addActionListener(new NewRound());
        menu.add(menuItem);
        menuItem = new JMenuItem("Close", KeyEvent.VK_Q);
        menuItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_1,
ActionEvent.ALT_MASK));
        menuItem.addActionListener(new FinishGame());
        menu.add(menuItem);
        frame.setJMenuBar(menuBar);
        range1 = new JRadioButton("0 <= X < 100");
        range2 = new JRadioButton("0 <= X < 5000");
        range3 = new JRadioButton("RESET");
        range3.setVisible(false);
        ButtonGroup group = new ButtonGroup();
        group.add(range1);
        group.add(range2);
        group.add(range3);
        range1.addActionListener(new NewRound());
        range2.addActionListener(new NewRound());
        range3.addActionListener(new NewRound());
        JPanel rangePanel = new JPanel();
        rangePanel.setPreferredSize(new Dimension(400, 70));
        rangePanel.setBorder(BorderFactory.createTitledBorder("Range
Options"));
        panel.add(rangePanel);
        rangePanel.add(range1);
        rangePanel.add(range2);
        rangePanel.add(range3);
        frame.pack();
        frame.setVisible(true);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
# method name: generateRandomNumber
! modifiers: 1
! type: void
! parameter: null
! body-blockstmt: {
    if (range1.isSelected()) {
        RandomNum = (int) (Math.random() * 100);
        range3.setVisible(true);
    } else if (range2.isSelected()) {
        RandomNum = (int) (Math.random() * 5000);
        range3.setVisible(true);
    } else {
        RandomNum = (int) (Math.random() * 1000);
        range3.setVisible(false);
    }
```

```
}
```

The output of the body statements line by line:

```
^^^^^^^^^^^^^^ if (range1.isSelected()) {
    RandomNum = (int) (Math.random() * 100);
    range3.setVisible(true);
} else if (range2.isSelected()) {
    RandomNum = (int) (Math.random() * 5000);
    range3.setVisible(true);
} else {
    RandomNum = (int) (Math.random() * 1000);
    range3.setVisible(false);
}


# method name: NewGame
! modifiers: 1
! type: void
! parameter: null
! body-blockstmt: {
    generateRandomNumber();
    number.setText("Guess a number");
    count = 0;
    continueBtn.setText("Continue Game");
    msgFrame.setVisible(false);
    countMsg.setVisible(false);
}
```

These are methods in the inner classes above (as the property of Swing library):

```
# method name: actionPerformed
# method name: actionPerformed
# method name: actionPerformed
# method name: actionPerformed
```

In Java, a frame represents a window of a program.

```
PModels :
JFrame  msgFrame
JPanel  btnPanel
```

```
JFrame  frame
JPanel  panel
JPanel  msgPanel
JPanel  rangePanel
```

Describe each widget of the example application's UI and where it refers to, and also list all of variables including global variables used in the program:

```
panel On frame.getContentPane()
Variables used in the methods: panel
label1 On panel
Variables used in the methods: label1
number On panel
Variables used in the methods: number
go On panel
Variables used in the methods: go
msgPanel On msgFrame.getContentPane()
Variables used in the methods: msgPanel
msg On msgPanel
Variables used in the methods: msg
Box.createRigidArea(new Dimension(0, 5)) On msgPanel
Variables used in the methods: Box.createRigidArea(new Dimension(0, 5))
exitBtn On btnPanel
Variables used in the methods: exitBtn
Box.createRigidArea(new Dimension(10, 0)) On btnPanel
Variables used in the methods: Box.createRigidArea(new Dimension(10, 0))
continueBtn On btnPanel
Variables used in the methods: continueBtn
btnPanel On msgPanel
Variables used in the methods: btnPanel
Box.createRigidArea(new Dimension(0, 5)) On msgPanel
Variables used in the methods: Box.createRigidArea(new Dimension(0, 5))
countMsg On msgPanel
Variables used in the methods: countMsg
menu On menuBar
Variables used in the methods: menu
menuItem On menu
Variables used in the methods: menuItem
menuItem On menu
Variables used in the methods: menuItem
range1 On group
Variables used in the methods: range1
```

```
range2 On group
Variables used in the methods: range2
range3 On group
Variables used in the methods: range3
rangePanel On panel
Variables used in the methods: rangePanel
range1 On rangePanel
Variables used in the methods: range1
range2 On rangePanel
Variables used in the methods: range2
range3 On rangePanel
Variables used in the methods: range3
```