Copyright is owned by the Author of the thesis. Permission is given for a copy to be downloaded by an individual for the purpose of research and private study only. The thesis may not be reproduced elsewhere without the permission of the Author.

School of Engineering and Advanced Technology

Adding Traceability to an Educational IDE

A thesis presented in partial fulfilment of the requirements for the Master degree in Computer Science at Massey University, Manawatu, New Zealand

Author:

Li Sui

Supervisor:

A/Pro Jens Dietrich

A/Pro Eva Heinrich

August 12, 2016

Contents

1 Introduction 9

2 Background 14

2.1 Challenges in teaching/learning programming and existing approaches 14

2.1.1 Gamification 14

2.1.2 Game classification 16

2.1.3 Existing educational platforms 17

2.2 Related work 21

2.2.1 SoGaCo 22

2.2.2 PrimeGame 23

2.2.3 PrimeGame strategy classification 25

2.3 Conceptual foundations 30

2.3.1 Notional machines 30

2.3.2 Conceptual model 32

2.4 Technical foundations 34

2.4.1 Continuations 34

2.4.2 The Java debug interface 35

2.4.3 Instrumentation libraries 36

2.4.4 Instrumentation 38

3 A Layered and Reversible Notional Machine 44

3.1 Bi-directionality 44

3.2 Two level hierarchy 45

4 Design and Implementation 48

4.1 Client 48

4.2 Server . . . 51
4.2.1 Snapshots . . . 51
4.2.2 Implementing tracing using source code instrumentation . . . 53
4.2.3 Implementing tracing using byte code instrumentation . . . 54
4.2.4 Discussion . . . 57
4.2.5 Compression methods and encoding schemes . . . 58

5 Experiment Analysis **65**
5.1 Methodology . . . 65
5.1.1 Build performance measurements . . . 65
5.1.2 Runtime performance measurements . . . 66
5.2 Build performance (source code instrumentation) . . . 67
5.3 Build performance (byte code instrumentation) . . . 69
5.4 Runtime performance . . . 71

6 Conclusion and Future work **78**
6.1 Conclusion . . . 78
6.2 Future work . . . 79
6.2.1 End user validation . . . 79
6.2.2 Compression improvement vs Game play strategy . . . 80
6.2.3 Other games . . . 80

7 Appendix **82**
7.1 Java Debug Interface Code Example . . . 82
7.2 JavaFlow Code Example . . . 87
7.3 ASM byte code . . . 88
7.4 BlackMamba . . . 103
7.5 Java Parser . . . 108

List of Figures

1 Thesis Structure . . . 12
2 Greenfoot Interface . . . 19
3 If condition in Blockly . . . 20
4 PrimeGame Board . . . 24
5 Cautious VS Greedy . . . 33
6 Prime number VS Cautious . . . 33
7 Java Platform Debugger Architecture [29] . . . 35
8 Example Code before Instrumentation . . . 39
9 Abstract Syntax Tree . . . 40
10 Example Code after Instrumentation . . . 41
11 Eclipse Debugger . . . 44
12 Program code Comprehension: Assignment . . . 45
13 Program code Comprehension: Method Call . . . 46
14 Game strategy comprehension: State Changing . . . 46
15 Visual plus texture view . . . 47
16 Editor page . . . 49
17 Testing page . . . 49
18 Bot Selection . . . 50
19 Choose Who Plays First . . . 51
20 Snapshot . . . 52
21 Method Flow of Source Code Instrumentation . . . 53
22 Nested Map data structure for Source Code Instrumentation . . . 54
23 Byte code Instrumentation . . . 55
24 Example code: Miss capture on different JDK version . . . 57
25 Level of Access: depth 2 . . . 59

26 LinkedList structure 60
27 Baseline Encoder Structure 61
28 Memory usage: Build Bots with source code instrumentation(MB) . 68
29 Memory usage: Build Bots without instrumentation(MB) 69
30 Memory usage: Build Bots Byte code instrumentation(MB) 70
31 Producing extra contents 76

Listings

1 Randomly strategy . . . 26
2 Cautious strategy . . . 26
3 Greedy strategy . . . 27
4 Largest prime number strategy . . . 27
5 Max-gain strategy . . . 28
6 For loop . . . 30
7 Debug For loop . . . 31
8 Bad Writing Habit . . . 43
9 A variable that stays the same across all the moves . . . 62
10 A variable that stays the same in a single move . . . 62
11 Changing variable . . . 62
12 Adding a variable using edit distance . . . 63
13 Remove a number in an array using the edit distance . . . 64
14 Remove a number in an array using the tree edit distance . . . 64
15 Monitor field . . . 82
16 Connect to VM . . . 84
17 Monitored Code . . . 86
18 JavaFlow Code Example . . . 87
19 (JDK version: 1.8.0_60)Bytecode Demonstration . . . 88
20 BlackMamba Source Code . . . 103
21 JavaParser Code Example . . . 108

List of Tables

1 List of educational platforms 17

2 Local Variable Table 55

3 Time for Building bots using source code instrumentation (milliseconds) 67

4 Time for Building bots using byte code instrumentation(milliseconds) 70

5 Different Encoding and Compression Result in Time(milliseconds) . 71

6 Different Encoding and Compression Result in Memory(KB). (blue:depth 2, red:depth 3) 72

Abstract

High dropout and failure rate in introductory programming courses indicate the need to improve programming comprehension of novice learners. Some of educational tools have successfully used game environments to motivate students. Our approach is based on a novel type of notional machine which can facilitate programming comprehension in the context of turn-based games. The first aim of this project is to design a layered notional machine that is reversible. This type of notional machine provides bi-directional traceability and supports multiple layers of abstraction. The second aim of this project is to explore the feasibility and in particular to evaluate the performance of using the traceability in a web-based environment. To achieve these aims, we implement this type of notional machine through instrumentation and investigate the capture of the entire execution state of a program. However, capturing the entire execution state produces a large amount of tracing data that raises scalability issues. Therefore, several encoding and compression methods are proposed to minimise the server work-load. A proof-of-concept implementation which based on the SoGaCo educational web IDE is presented. The evaluation of the educational benefits and end user studies are outside the scope of this thesis.

Acknowledgments

Firstly, I would like to express my sincere gratitude to my supervisor A/Prof. Jens Dietrich for the continuous support of my study and related research, for his patience, motivation, and immense knowledge. His guidance has pointed me in the right direction throughout the work.

I also would like to express my appreciation to my co-supervisor A/Prof Eva Heinrich for her patient guidance and advice on computer science education. My sincere thanks also goes to Prof. Manfred Meyer and Mr. Johannes Tandler for their active collaborations on writing related papers.

Finally, I must express my very profound gratitude to my parents with unfailing support and continuous encouragement.

Introduction

This study is based on a number of research problems in the area of computer programming education. First of all, the motivation in learning programming language. Programming courses have a high dropout and failure rate [51] [9] [22]. One possible approach to improve the output of programming and introductory computer science courses is the provision of dedicated educational programming environments. These environments aim at lowering the threshold for entry for students by using only a reduced feature set compared to industry strengths integrated environments (IDEs). Secondly, the use of the visualisation addresses the fact that "novice learners can become quickly overwhelmed by too many details, and they usually prefer to test an animation with predefined input data" [41, p. 132]. These environments often use some level of gamifications to motivate and engage students. Several such environments have been designed and used in recent years, including Greenfoot [31], Blockly [23]. and Robocode [27], and there are some evidences that the use of such tools can indeed improve educational outcomes [5] [43]. These environments promote programming and program comprehension in a way of gaming, visualising and traceability. Thirdly, despite the benefit that are using above techniques to improve programming comprehension, some of client-based environments like Greenfoot and Robocode have high total cost of ownership as they incur per client maintenance costs. On the other hand, Blockly is web-based visual programming editor but does not support traceability.

Our work is based on a platform called Social Gaming Coding (SoGaCo) [18]. A unique feature of SoGaCo is that it is a web-based platform and also provides a Software as a Service (SAAS) [28] like deployment model with a HTTP-based API [21]. This architecture addresses several requirements: (1)It is a cloud-based

platform and can be deployed as a service to ensure low total cost of ownership for organisations (such as high schools); (2) It facilitates the integration of services such as identity management (through providers like Google, Facebook or in-house providers via standards such as LDAP[1]); (3) It facilitates social networking features for users to interact, in particular in order to develop programs of "bots" that compete in tournament-like settings. SoGaCo as a framework supports multiple programming languages and different games. We are mainly using Java as programming language, and the PrimeGame [39] as a game. The PrimeGame is a simple mathematical board game that has been successfully used in computer science education for many years.

This study aims to improve program comprehension by introducing notional machine [13]. The idea of notional machine is to expose students to models showing the program execution. A notional machine is an idealized, conceptual computer whose properties are implied by the constructs in the programming language employed [13]. The purpose of a notional machine is to offer support for interpreting the behaviour of running programs. The novel contributions of this study are described in the following paragraphs.

Traditional techniques to provide notional machines are debugger-like tracing features, either backed by an actual debugger API of the platform like JDI [29] or by a continuation language feature such as JavaFlow [17].

Because the cost of maintaining active sessions between the client and server is very high in term of memory, traditional debuggers do not work well in a web-based environment where code is executed on the server. We have therefore designed a

[1]The Lightweight Directory Access Protocol

new type of notional machine that:

- Can be embedded seamlessly into the web-based environment;
- Supports bi-directionality by facilitating “reverse debugging”;
- Uses a two-layer approach that uses a domain-specific visualisation using a game board combined with a code view to facilitating program comprehension.

This design is not straightforward as it requires us to capture the entire state of program executions on the server, marshal it and transfer it to the client for visualisation. In particular, it is not apparent that this design scales as it consumes significant resources in terms of time, memory and network bandwidth. In this thesis, we try to quantify these aspects in order to evaluate the feasibility of such a design. We investigate the following variation points of this design:

- We monitor the time and memory usage to gather the state of an executing program through Java bytecode instrumentation.
- Different methods to encode and compress the results of collected data. These methods are baseline encoder, custom encoder, dictionary index method, edit distance method and tree edit distance.

We choose to use JMH[2], a micro-benchmarking tool that written in Java. It is targeting JVM for building, running and analysing benchmarks. We are using this tool for detailed analysis such as average, variance, standard deviation confidence intervals.

[2]http://openjdk.java.net/projects/code-tools/jmh/

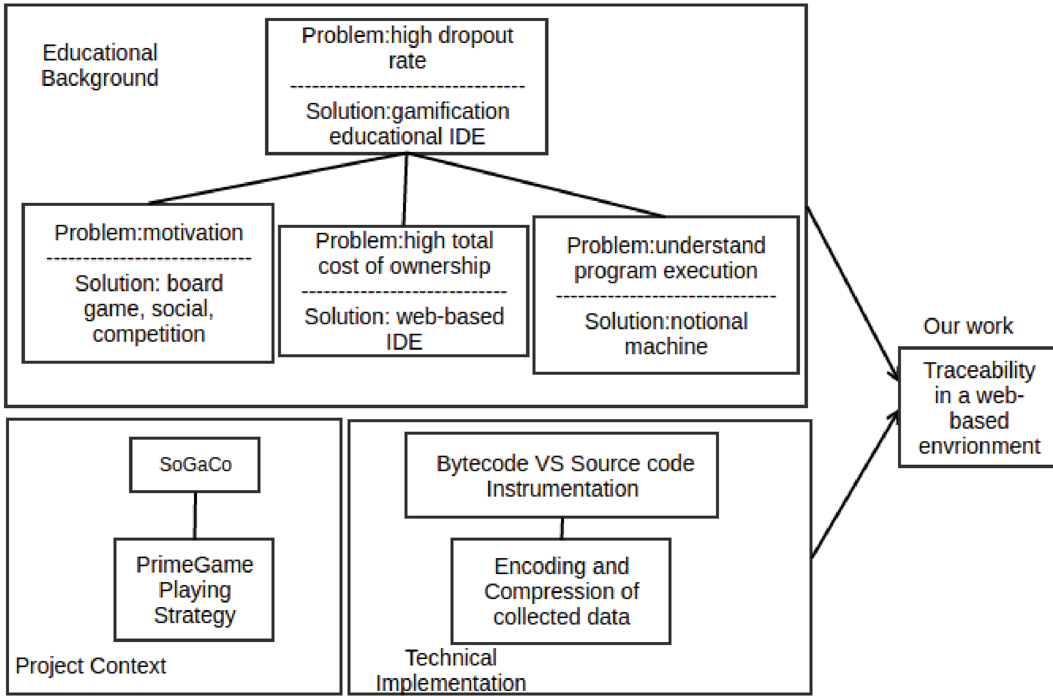

Figure 1: Thesis Structure

In summary,Figure 1 demonstrates the overview structure of this thesis. The thesis contains 3 parts. The first part covers the educational aspects. The gamification and educational IDEs are the two possible solutions for addressing the difficulties in learning program and the related high dropout rates. Traditional IDEs have a high total cost of ownership because it requires an independent client to be installed and maintained. The second part addresses that issue by presenting the SoGaCo, which is a low total cost of ownership in a web-based environment. SoGaCo also integrates games and social networking to improve the motivation for students. In the third part, we implement the traceability to facilitate program comprehension which inspired by a concept of the notional machine.In particular, we evaluate the use of different instrumentations to capture and gather states during program execution, and compression methods to minimise the server work-load

from collected data. Scalability is a critical aspect here due to the amount of data producing by the different PrimeGame playing strategies. The evaluation of the educational benefits and end user study are outside the scope of this thesis.

Background

Challenges in teaching/learning programming and existing approaches

In this section, we discuss the use of gamification, which has great influences on motivation. There are two types of game: video game and serious game. They are also discussed in this section based on their educational benefits. Particularly, we introduce the broad game as it can be described in states and recorded in an orderly fashion. That is the reason we choose to implement traceability in an IDE that consist multiple games. There are a number of educational platforms available in the market. They are all using games as a learning media. We analysis pros and cons of each one of them.

Gamification

There is a problem that "students nowadays will easily lose enthusiasm and interest in learning computer programming, especially when they experience repetitive failure in practising on their own" [33, p. 218]. To address this problem, our approach is based on the assumption that the use of games to teach programming motivates the novice learner and therefore improves educational outcomes. Analysing motivation can help us to explore the educational potential of using games. In general, there are two types of motivation [34].

- Extrinsic. This can be described as the external aspect of motivation. For example, rewards related to career opportunities is an extrinsic motivation for programmers entering the industry.
- Intrinsic. This can be described as the internal aspect of motivation. The

primary motivator is spontaneous interests in learning how to program.

The demand for programming skills has grown almost 50% since April 2002 [45]. More and more educational institutions are well aware of the importance of teaching programming skills and the strong job market. However, this demand does not necessarily help to improve students intrinsic motivation for learning programming especially when it comes to very technical aspects. Games may provide a suitable platform using game-based learning theory [46] [60] to improve motivation. In fact, games can motivate students to learn, improve their self-esteem, minimise the time needed for knowledge acquisition and also reduce instructor's load [50] [30]. Examples of the use of gamification in programing education include Greenfoot [31], Blockly [23] and Robocode [27]. The work by Rajaravivarma and Rathika suggests that the use of game could potentially benefit computer education in areas like problem solving and logical thinking [47].

A number of empirical studies [44] [58] [11] show the effectiveness of game-based learning. For example, one study [58] compares game-based applications with non-game-based applications with respect to the learning effectiveness and the motivational appeal. In this study, two groups of high school students were invited to learn computer memory concepts. One group used a game-based application (Group A) and the other group used a non-gaming application (Group B). A computer memory knowledge test was taken by the students afterwards. The results show a statistically significant effect on scores in favour of Group A, which indicates that students that had used the gaming application (Group A) performed significantly better in the test than those that had used the non-gaming one (Group B).

Game classification

Beregeron distinguished between video games and serious games for programming education [10]. Video games provide more visual aids and effects. On the other hand, serious games, can enhance the acquisition of knowledge and cognitive skills [62] [8].

Board games are one of the sub-category of serious games. The SoGaCo platform we use and discussed in more detail below (Section 2.2.1), contains several board games, such as PrimeGame, Mancala and Othello. One of the characteristics of board games is that they are turn-based. More details will be explained in Section 2.2.1

State is an important concept to understand computation. A program consists of multiple states and a turn-based game consists of multiple turns. A turn-based game describes not only the state of game but also the state of program execution. On the other hand, video games use complex user interfaces that often obfuscate state. Compared to turn-based games, programming real-time video games can be considerably more complex than programming board games due to the unpredictable game play condition [19]. To work with video games, students need knowledge and skill of sound effects, networking and animation, and this is challenging [35]. Distraction is another reason to chose turn-based (serious) games over video games. Too much visualization could cause massive distraction which could lose education benefits [41]. Our aim is to create an environment where students focus on solving programming problems like algorithm design rather than the game itself. In this case, the game acts like an agent to assist students to achieve this goal.

In general, we believe that the serious game's inherent properties promote strategic thinking and their inherent simplicity are more important than the video game's exciting fast-paced interaction and sophisticated immersive multimedia in programming education [19].

Existing educational platforms

In this section, we analyse a number of platforms available. We focus on some unique features which have inspired our own design.

Table 1 shows a number of educational platforms. Program visualization is a common feature for all these platforms. However, the language they support and the platform they are running are different, and choices take targeted audiences into account. Platforms like Scratch[3], Blockly [23] and Codecombat[4] support a graphical programming language which put emphases on teaching concept of program. Greenfoot, Robocode and BlueJ[5] have clear targeted teaching language but the rest of platforms shown in the Table 1 only support a graphical programming language. Traceability is only supported by standalone platforms, as shown in the Table 1. Web-based systems usually does not include a debug features. This is because a traditional debugging mechanism is not easy to support in a web-based environment. This issue leads us to design debugging mechanism in a web-based environment.

Table 1: List of educational platforms

Platform	Suitable for	Teaching language	IDEs	Debugging

[3] https://scratch.mit.edu/
[4] https://codecombat.com/
[5] http://www.bluej.org/

Greenfoot	high school and early university level	Java	standalone	Yes
Alice	children	graphical programming language	standalone	Yes
Robocode	university level	Java/C#	standalone	Yes
Codecombat	beginner and gamer	graphical programming language	web-based	No
Scratch	children	graphical programming language	web-based	No
Blockly	beginner	graphical programming language	web-based	No
CodeMonkey	all age	Coffeescript	web-based	No
BlueJ	introductory university course	Java	standalone	Yes

In the following paragraphs we describe 3 platforms that inspire us designing our platform in more details.

Greenfoot

Greenfoot's target user group is around 14 years of age, but it can also be used for introductory programming courses at university level [31]. Greenfoot does not use a static main method as a program entry point. It consists of three elements

of state which are represented visually: a rotation, a position, and an image [31]. An animation effect is generated by the corresponding methods. For example, the method `setRotation(60)` can rotate an object to 60 degrees and `setLocation(5, 3)` can set an object at a specific location. Synchronicity is another unique feature of Greenfoot. Once some classes have been compiled, the context menu can display the corresponding classes as Figure 2 shows.

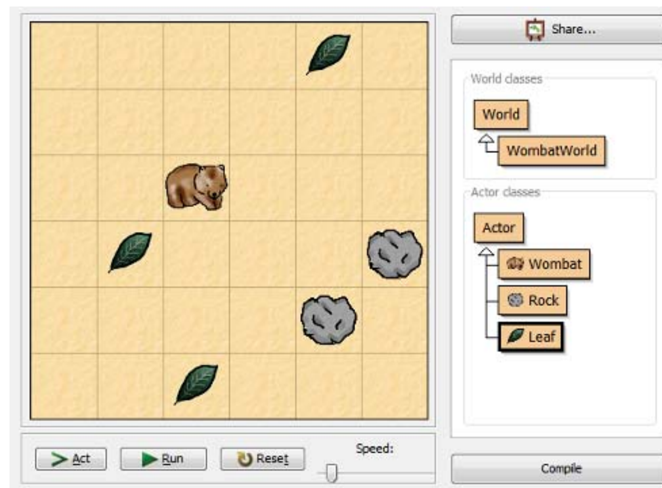

Figure 2: Greenfoot Interface

The limitations of Greenfoot include error handling and reporting. The messages used to report errors are often not especially helpful and learners could be frustrated by the high learning curve for interpreting error messages [31].

Blockly

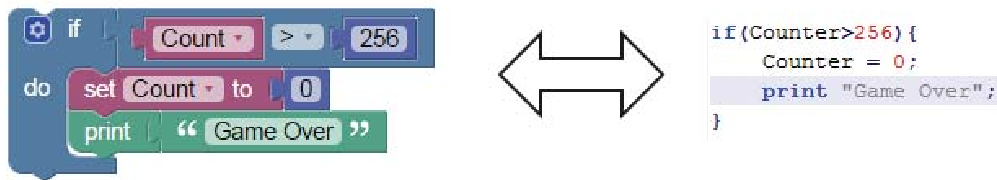

Figure 3: If condition in Blockly

s

Blockly is a web-based educational platform for teaching introductory programming [23]. It is suitable for beginners at all levels, including children. Blockly adapted the concept "fun to use" instead of "learn to use", and this makes it becomes a very user-friendly learning platform [38]. Unlike Greenfoot, Blockly game design uses certain tasks for certain games. For example, solving a maze problem requires students to understand conditional and loops. Blockly also helps the users to simplify program models in order to make it comprehensible. For example, the Figure 3 shows how visual codes is associated with traditional code. Blockly focuses on intrinsic motivation for novice learners by providing friendly user interface and comprehensible model design. Blockly also provides mapping functions to convert a graphical programming language to an actual programming language like Javascript. However, the Blockly's simplicity is also its limitation as it makes it difficult to use for more complex problems.

One of the uses of Blockly is MIT App Inventor 2[6]. It has been developed by Google and maintained by the Massachusetts Institute of Technology (MIT). MIT Inventor 2 is used for developing android applications. The ready made component and event handlers in MIT Inventor provide a drag and drop function to facilitate

[6]http://ai2.appinventor.mit.edu/

the development. For example, users can put a ball or image-based object on a canvas.

Robocode

Robocode is designed for learning programming by developing a robot battle tank to against other battle tanks. It supports not only Java but also .NET. The interoperability of Robocode provides a possibility for learning programming on the same platform despite what programming languages they are familiar with. The robot battles are running in real-time. Robocode has a standalone development environment. It has its own installer, editor and Java compiler. Robocode also can be supported by external IDEs such as Eclipse. Robot API provides several functions to control bots such as movements and events. As a teaching tool, Robocode is currently used in some introductory courses [26]. It also facilitates some standard pedagogical methods, particularly for modeling objects in game environments and problem based learning (PBL) [61] in a context of competition. [43]. One important feature of Robocode is that it provides a multi-user environment to support competition. Users can upload their bot to a server and run it against other bots. The idea of competition is to motivate students to engage in a social/interactive environment [27]. The Robocode national competition offers open-ended problems. It requires participants designing the strategy differently on the different stage of competitions or facing different opponents. They collaborate with each other in a small group to discuss the strategy, and then try to solve the problems together.

Related work

In this section, we describe the previous works that our project is based on.

SoGaCo

SoGaCo[7] platform is a start point for our work to build on. SoGaCo is a scalable web environment that supports multiple programming languages to build competitive bots, which play simple mathematical board games [18].

SoGaCo facilitates the idea of the problem-based learning (PBL). In fact, game is a perfect platform for PBL [42] [52]. In this method, SoGaCo provides a teaching platform for students to explore programming languages and different game strategies. When students are playing board games, each player takes a turn to continue the game according to a set of rules. This property of board games can break down the algorithmic complexity of the game into multiple stages. This the advantage of understanding how the program that plays the game runs by computing each turn in separation.

One of the unique features of SoGaCo is social networking. Social networking features are believed to be very popular with students. It facilitate the delivery of new knowledge by sharing information. Social networking is also an efficient way to intrinsically motivate students to learn [48], [12]. SoGaCo takes advantage of this by including competition and collaboration in order to achieve better educational outcomes. The sense of participation in a competitive environment can improve students' interests [15]. Students can program their own programs ("bots") and share them by sharing the unique bot address (URL) through emails or other social networking media, inviting others to compete with this bot by loading (clicking) this URL. This could be taken further by organising on-line tournaments. This has not yet been implemented in SoGaCo, but it is a feature that is planned.

[7] http://sogaco.massey.ac.nz/

In the SoGaCo platform, bot building is the process that turns source code into bot objects [18]. For the Java language, this includes 6 parts: compilation, security check, instrumentation, loading, instantiation and testing.

- Compilation: Compile source code into byte code using a corresponding compiler.
- Security check: To prevent injection attacks, a byte code check against an API white-list of permitted classes is required.
- Instrumentation: The code is instrumented for monitoring the timeout and resource usage. This is to enforce the security by preventing denial of service attacks using resource (memory and CPU) quota.
- Loading: Load the Java class that is generated by the compiler.
- Instantiation: The class is instantiated.
- Test: JUnit acceptance test is to detect subtle problems like illegal moves (postcondition violations), slow computations (via timeouts) and runtime problems such as stack overflow errors.

PrimeGame

SoGaCo has a modular architecture that supports different games to be "plugged-in". One such game currently supported is the PrimeGame. The PrimeGame is a simple mathematical board game developed by Meyer et al [39]. Its rules are very simple: the board initially consists of numbers from 1 to 100, and in each turn, the player selects a number from the board that has not yet been played. This player then gets this number added to his or her scores. However, the opponent gains all factors of this number that are still on the board.

The structure of PrimeGame can be easily represented visually. An array of 100 numbers from a 10*10 game board as shown in Figure 4. Different colors demonstrate the status of PrimeGame. For example, the dark red is the numbers that player 1 played and the dark blue represent player 2. The light colors are the number scored through a move made by the opponent.

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Figure 4: PrimeGame Board

This representation provides an intuitive visualization of the internal states within a running program and facilitates step by step comprehension.

The aim of using the PrimeGame is to train students to devise more and more sophisticated strategies. For instance, assume player 1 starts the game and plays the largest number available, 100. Therefore, player 1 will gain 100 points. However, according to the rules, player 2 will get all the factors of 100: 1,2,4,5,10,20,25 and 50, this is 117 points! A better strategy would be to play 97. Then player 2

would only get 1 point as this is the only factor of 97. However, implementing this strategy requires the understanding of more programming concepts such as loops, conditions and inter-procedural calls.

PrimeGame strategy classification

In order to increase the novices programmer/students' competence, it is required to not only become proficient in a certain programming language but also to have a good understanding of algorithms and problem solving. Von Mayrhauser and Vans [57] described an important characteristic of being an expert programmer: the ability to apply both general problem solving strategies and specialised strategies. In SoGaCo, designing programmable and competitive bots require students to employ different algorithms and data structures corresponding to different playing strategies. In this section, we present a number of game playing strategies for the PrimeGame. We rank them from low level to high level based on its complexity.

Because students have different levels of understanding and different background knowledge, educators must be cautious to choose the appropriate level of difficulty for teaching. We have already discussed the differences between video games and serious games (Section 2.1.2), but we have not classified anything in more detail. Now we propose a classification based on the PrimeGame playing strategies, associated computational costs, programming efforts and level of proficiency needed to implement those strategies.

1. low level/simple: A simple strategy is driven by the low complexity of the implementation. The objective is for bots to work correctly regardless of their strength. Here are three examples of low/simple strategies.

(a) Random strategy

Randomly select a position to move as long as the move satisfies the game rule. The choice taken is strongly influenced by the simplicity of the code required to make this choice. For example, in the PrimeGame, pick a random valid number at each turn (Listing 1).

Listing 1: Randomly strategy

```
@Override
public Integer nextMove(List<Integer> game){
   Random r = new Random();
   int pick = r.nextInt(100);
   return game.get(pick);
}
```

(b) Cautious strategy

The cautious strategy always pick up the first number until the numbers run out (Listing 2).

Listing 2: Cautious strategy

```
@Override
public Integer nextMove(List<Integer game>){
     return game.get(0);
}
```

2. medium level: Medium level is a locally optimised strategy.

(a) Greedy strategy

For the game where the player gains points counting toward an overall win by points during each move, a greedy strategy tries to capture as

many points as possible during each turn. In general, a greedy strategy can be described as optimised for the current game state. For example, one of the PrimeGame strategies is to gain maximum points at each stage. As consequences, picking the largest number from the list of possible numbers is a best solution for the greedy strategy (Listing 3).

Listing 3: Greedy strategy

```
@Override
public Integer nextMove(List<Integer game>){
    int index = game.size()-1;
     return game.get(index);
}
```

(b) Prime number

This strategy requires to find the largest prime number in the list.

Listing 4: Largest prime number strategy

```
@Override
public Integer nextMove(List<Integer> game) {
   int largest = 0;
   for(int i=0;i<game.size();i++){
      if(isPrime(game.get(i)) && game.get(i)>largest){
         largest = game.get(i);
      }
   }
   return largest;
}
private boolean isPrime(int n){
   for(int i=2;2*i<n;i++) {
```

```
            if(n%i==0)
                return false;
        }
        return true;
    }
```

(c) Max-gain

Max-gain strategy (Listing 5)secures maximum scores in one round relative to the score gained by the opponent. So this is the "net gain". We refer to this as SmartBot later on.

Listing 5: Max-gain strategy

```
@Override
    public Integer nextMove(List<Integer> game) {
        Set<Integer> numbers = new HashSet<Integer>();
        numbers.addAll(game);
        int selection = -1;
        int maxGain = Integer.MIN_VALUE;
        for (int n:game) {
            int gain = n;
            for (int i=1;2*i<n;i++) {
                if(n%i==0 && numbers.contains(i)) {
                    gain = gain - i;
                }
            }
            if (gain>maxGain) {
                selection = n;
                maxGain = gain;
            }
```

```
    }
    return selection;
  }
```

3. High level: High level strategies are exhaustive uses heuristics. This implies a high complexity. Some of the strategies are not suitable for PrimeGame since it is a simple mathematical game.

   (a) Look-ahead strategy

   Look-ahead strategy. The next move is chosen not only based on the current state, but also on a prediction of the next moves. An example called BlackMamba is provided in Appendix 7.4

   (b) More advanced strategy

   This allows inspecting the whole decision tree to determine a move. The associated computation cost is high to very high. Possible tactics tricks could be applied. For example: set up a luring move to let opponent step into traps. Opponent modeling can be seen as a classification problem, where an opponent is classified as one of a number of available models based on data that is collected during the game.

This classification provides educators wide range of choices for different students. Moreover, educators could use this feature for assessment. Competing against built-in programs based on different levels can generate 3 results: win, lose or draw. These results give educators a standard for marking students' assignments.

Conceptual foundations

In this section, we briefly introduce the concept of notional machine.

There is a number of studies [32] [54] that have revealed a major issue with the way programming is taught. For instance, a survey was conducted for 2nd year programming students, and lecturers and teachers of programming from around the UK [40]. Researchers found out that a major problem was that students did not understand what happened when the program is executing. For example, beginner students are confused about the following questions: how did the variables change at a certain stage and what does an algorithm do in general? A possible solution is using a debugger to retrieve program states which demonstrate how it runs. Moreover, we expect the visualisation for program models to promote the semantic interpretation that a debugger can provide. In the following sections, we introduce two ways to assist program comprehensions.

Notional machines

The following Listing 6 is written in Java demonstrating a `for` loop that check if it is a prime number.

Listing 6: For loop

```
for (int i=0;i<5;i++) {
  isPrime(i);
}
```

Novice learners prefer to interpret a program line by line rather than using meaningful program chunks on structure [51]. A reasonable description of that

`for` loop is: “declare an integer i, assign it with 0, and it will iterate over all the values from 0 to 4 and check if it is a prime number”. However, the semantics of a `for` statement is mixed up with what the `for` statement is used for in the particular context [53]. For example, Vainio’s study [55] shows some students think the variable within a `for` loop is always set to zero at the start of the loop. Thus, it is important to demonstrate the actual program execution step by step (Listing 7). Du Boulay introduced the idea of the notional machine and referred to it as ”the general properties of the machine that one is learning to control as one learns programming” [13]. It is an abstract computer machine whose properties are implied by the constructs in the programming language employed [13]. He argues that a notional machine should be simple, useful and can be used to observe the inside of a program model. To use a metaphor: a glass box should do better work than a black box in terms of explaining program execution. According to these issues addressed above in teaching programming, the notional machine is a suitable theoretical framework for us. We apply traceability to establish observation on how program runs internally.

Listing 7: Debug For loop

```
stage1: isPrime(0);
stage2: isPrime(1);
stage3: isPrime(2);
stage4: isPrime(3);
stage5: isPrime(4);
```

To summarize the characteristics of notional machine [53]:

- It is a representation of a program at runtime;
- It serves the purpose of demonstrating what is going on program execution;

- It describes the semantics of program code which is written in those paradigms or languages (or subsets thereof)

Conceptual model

The conceptual model crafts the system structure and presents it in a high level of abstraction [59]. The ideas of the conceptual model is to improve the program comprehension by visualising the program state. For example, the board of PrimeGame in the SoGaCo (Figure 4) is a representation of the PrimeGame'state. People can tell what number been taken from the game board corresponding to what number been taken from a list. If the number which display in a game board is not what people want , then there must be something wrong with manipulation of the list in the source code. Another example of using conceptual model is to demonstrate the strategic awareness though game broad. Figure 5 shows that the red bot is using the cautious strategy (Listing 2) which only chooses the smallest number on the board and the blue bot is using the greedy strategy (Listing 3) which picks the largest number on the board. The respective student can clearly see that the red bot has an overwhelming advantage because the blue bot chooses a larger even number which gives the red bot more points due to the prime factor rule.

Figure 5: Cautious VS Greedy

On the other hand, the red bot could adapt a better strategy (Figure 6), for example picking up the largest prime number. Then, the blue bot(cautious strategy) would not gain extra points.

Figure 6: Prime number VS Cautious

The difference between two game play strategies can be clearly seen in Figure

5 and Figure 6. This gives students an overall situational awareness. The conceptual model can enable students to quickly comprehend the shortcomings and advantages of certain strategies.

Technical foundations

In this section we discuss several possible technologies that could be used to implement a notional machine which can describe the state of an executing program.

Continuations

Continuation was first discussed by Adriaan van Wijngaarden in 1964 [56] in a context of program's transformation into continuation-passing style [49]. With the continuation, an application program can capture the execution state of a program at a certain point. It usually involves the call stack which stores the information about a program's subroutines. A successful example of how continuation can be used in object-oriented programs is Seaside [20], a server-side web application framework written in Smalltalk. Continuation is used to model complex sessions, where the execution of a method implementing the session is interrupted in order to interact with the web client. This facilitates the use of the procedural logic of the programming languages to model the logic flow within web application sessions.

Unfortunately, continuation is not directly supported by Java. However there is some support through libraries such as JavaFlow [17]. JavaFlow implements continuation via byte code instrumentation. There are two ways to instrument byte code. One way is to use the JavaFlow Ant Task to build and enhance classes with continuation support at build time [17]. Alternatively, a special class loader called ContinuationClassLoader can transform a class at runtime. A simple code

example is provided in Appendix 7.2.

One of the disadvantages of using continuation is its lack of scalability. Since our project is running in a web-based environment for multiple users with potentially many concurrent sessions, the use of continuation could lead to a significant amount of memory been used by server to maintain active sessions.

The Java debug interface

The Java Debug Interface (JDI) is a high level Java API providing information useful for debuggers and similar systems needing access to the running state of a virtual machine. JDI is one layer of the Java Platform Debugger Architecture (JPDA) [29].

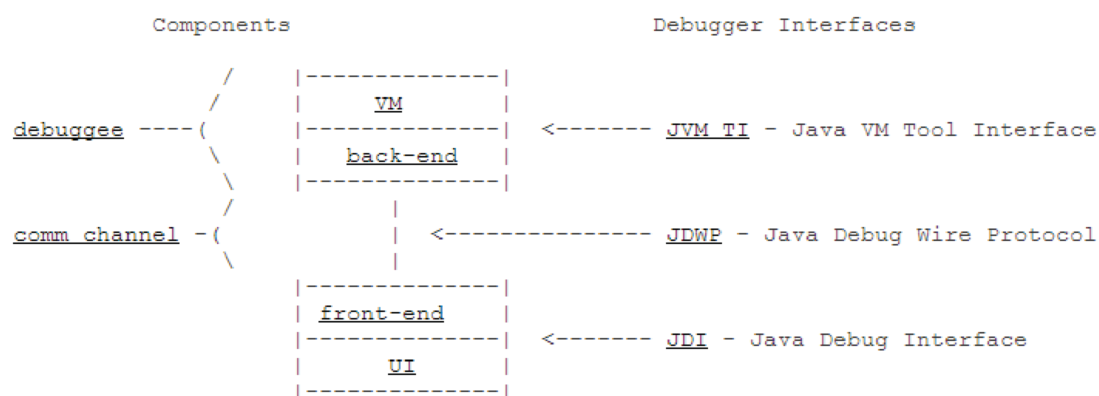

Figure 7: Java Platform Debugger Architecture [29]

As the Figure 7 shows JPDA consists a front end JDI, a communication channel called Java Debug Wire Protocol, and the back-end JVM that actually suspends a thread's state and allows access to local variables. Similar to JavaFlow, they are all able to pause at a certain state of a program, but JDI is embedded deeply into JVM. The application pauses as the JVM suspends on the stack. This can

cause a significant performance overhead when this technique is implemented on the server. A simple code example is provided in Appendix 7.1.

In summary, we found traditional debugging technique is not suitable for our approach for several reasons:

- In a web-based setting with server-based execution, this approach leads to open sessions that can consume significant amounts of memory, and therefore compromise the scalability of the platform;
- Debugger-like tracing is inherently directional, and in order to go back in history, the user has to restart execution. This is cumbersome and not intuitive;
- Debugger-like tracing is not only code-centric, but also a more intuitive notional machine that takes advantage of the domain-specific visualisation of games;

Instrumentation libraries

SUN compiler API

The SUN compiler library[8] is developed by the Oracle. The SUN Compiler API uses the visitor pattern [24] to visit through all different types of statements and expressions. But this library is not designed to rebuild or modify the AST. By using StringBuilder, we parse out source code and insert statements based on line number which retrieved from AST visitor. More details will explained later in Section 4.2.2.

[8] http://docs.oracle.com/javase/7/docs/jdk/api/javac/tree/com/sun/source/util/package-summary.html

Oracle is working on Project Jigsaw[9]. It is designed to a standard module system for the Java SE Platform. Since Project Jigsaw will modularize the Java runtime, internal APIs (eg: packages com.sun *) will be made unavailable and the internal structure of the JRE/JDK will change, which includes folders and JARs. Following their deprecation in the Java 8, the endorsed standards override mechanism and the extension mechanism will be removed in Java 9. The class and resource files previously stored in lib/rt.jar, lib/tools.jar,lib/dt.jar [3], and various other internal jar files will now be stored in a more efficient format in implementation-specific files in the lib directory. Thus, maintainability of our project will be affected by this change.

JavaParser

JavaParser[10] is lightweight source code instrumentation library. It not only provides a AST parser but also allows to modify or create an AST from scratch. The current version is 2.3.0. However, JavaParser is protected under GNU Lesser General Public License. This would put some constraints on licensing our project. Another reason we did not choose this library is it is not constantly maintained. So, we are worry about the maintainability of it. A code demo is provided in Appendix 7.5 shows how to create compilation unit by using the JavaParser

ASM

ASM[11] is byte code manipulation framework that written in Java. It is also a visitor-based. Compared to above tools/libraries, ASM has following advantages [14]: Developers do not need to deal directly with a class constant pool and offsets within method byte code. It is focused on simplicity of use and perfor-

[9] http://openjdk.java.net/projects/jigsaw/
[10] https://github.com/javaparser/javaparser
[11] http://asm.ow2.org/

mance. ASM is well maintained and supported by a large user community.

BCEL

The Byte Code Engineering Library (BCEL) [7] is another tool to manipulate byte code. It is developed by Apache Foundation. Both ASM and BCEL are a low level access tool. They provide direct control of the byte code instructions. However, BCEL hasn't been updated for few years [1].

Javassist

Javassist (Java Programming Assistant) [16] is another byte code manipulation tool. Unlike ASM, Javassist does not require to work at byte code level. One way to profiling code is using the annotation to log a method which helps keep the code clean and simple. Beside, the audit logging can be removed without modifying the source code. Another way is using Java agent [2] which is a core Java feature introduced since Java 1.5 to manipulate the byte code. It hooks a pre-main method to register a class transformer before the JVM loads the actual class.

In summary, we choose the SUN Compiler API as the library for the source code instrumentation and ASM for the byte code instrumentation. Both of two libraries provide a handy AST like visitor which helps to extract the states of a program.

Instrumentation

This section presents two instrumentation methods. They are source code instrumentation and byte code instrumentation. We will later use instrumentation in order to capture the application state in order to implement a debugger-like function.

Code instrumentation is a mechanism which can modify original code for observing a running software's internal status [6]. Particular use cases of code instrumentation are measuring the program's performances and operation analyses. Instrumentation is also used to implement test coverage tools and aspect-oriented programming (AOP)

The Java compiler compiles source code into a Java class file which contains byte code. The concept of instrumentation is based on the abstract syntax tree (AST). It is a syntactic structure representation of a program. It is used to represent source code structure in an abstract level to facilitate analyses and processing. For example, an abstract syntax tree (Figure 9) demonstrates the following code (Figure 8).

```
while(b<4){
    if(a<b){
        a=1;
    }else{
        b=b+1;

    }
}
return a;
```

Figure 8: Example Code before Instrumentation

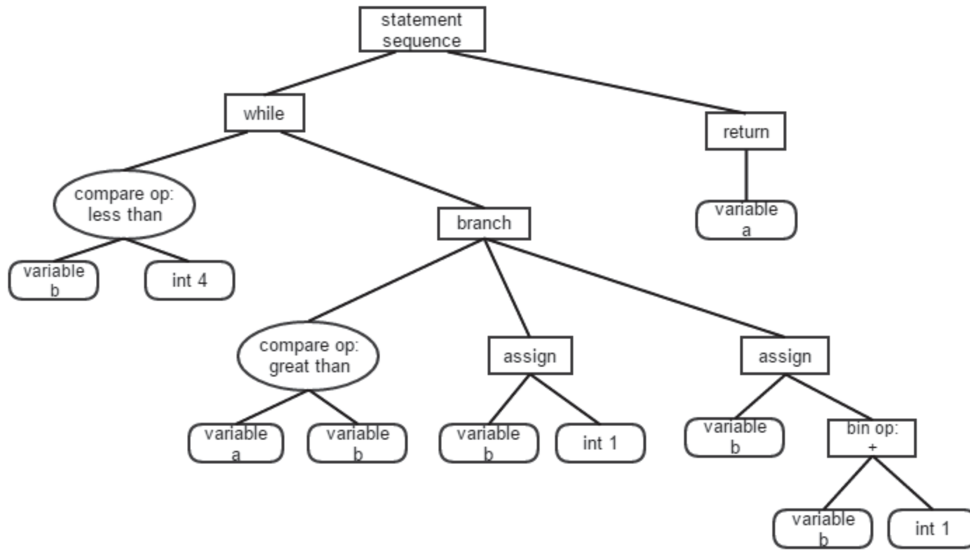

Figure 9: Abstract Syntax Tree

As we can see from above diagram (Figure 9), the AST provides direct access to each operation (node) and its operand (leaf). For retrieving proper tracing information, we only need to investigate the following operands and operations: classes, method invocations, fields and assignments. In this case, “a=1” and “b=b+1” need to be captured. Then, after instrumentation, the original code is modified and turned into following source code (Figure 10):

```
while(b<4){
    if(a<b){
        a=1;
        tracingEvent(a,b);
    }else{
        b=b+1;
        tracingEvent(a,b);
    }
}
return a;
```

Figure 10: Example Code after Instrumentation

To note that the “return” statement does have to be captured. In a circumstance that return a function like “return a+1;” does change the value of a, however it is not possible to capture this change after “return” in source code. To deal with the method invocations, we inject tracing event handler after every method invocation.

In this project, we apply source code and byte code instrumentation to capture the states of program.

Source code instrumentation

Source code instrumentation is an easy and intuitive way to instrumentate the program. It allows to inject extra code to source code. Compared to byte code instrumentation, source code instrumentation has following advantages [4].

- Flexible: Source code instrumentation allows to modify source code directly and insert code where appropriate. Usually this technique can be turn off/on since this takes place before compilation.
- Ease of use: Byte code is a condensed and optimized form of the original

source code. This could potentially cause trouble if developers are not familiar with the byte code. On the other hand, developers can easily understand what has been modified since it works on source code level.

- Source code can provide more information about the program. For instance, Java code: `public static final int A = 1`, is an example of constant inlining [25]. It sets the value of `A` is 1. However in byte code, `iconst_1` indicates integer value 1 been pushed to the JVM operand stack. It does not show the value 1 is obtained from `A`. Another example is using type erasure [25] to enforce the type safety in the Java run-time environment. So, in byte code, some generic types information has been removed.

Byte code instrumentation

Byte code instrumentation is based on the Java byte code. Byte code is set of instructions that exists in the Java virtual machine [36]. The Java source code is compiled into byte code and then the Java virtual machine loads and executes this byte code. When working with byte code, the Java source code is no longer required.

Unlike AST in source code instrumentation, byte code instrumentation deals with instruction sets. Java byte code instruction set can be grouped into following categories [36].

- Object control and initialization
- Operand stack control
- Arithmetic and logic
- Type conversion

- Logic control transfer
- Load and store
- Method invocation

The reasons we choose byte code instrumentation over source code instrumentation are listed below.

- Line matching. There is no line indication for the end of a condition. Poor code writing habits potentially cause line number mismatching. An example (Listing 8) shows that source code instrumentation picks up an assignment within a loop and register trace events right after it despite the fact that the loop condition is not well formatted.

  Listing 8: Bad Writing Habit

  ```
  for(int i =0;i<11
   addTrace();
  ;i++){
  }
  ```

- Byte code instrumentation is already used in the SoGaCo. An instrumented bot is used to detect the resource (memory and CPU) quota. To ensure the free memory is below configured percentage value or maximum memory, we measure the current memory use of all available objects. If memory usage exceeds, an exception will be thrown. So, we integrate the traceability with existing instrumentation to maintain consistency in our platform.
- Byte code is simpler to process as class names are resolved. In the Java source code, the package information is separated from where it is applied. But the byte code instruction includes the package information.

A Layered and Reversible Notional Machine

We already discussed the concept of notional machine (Section 2.3.1). For the purpose of revealing the program execution, we apply traceability to construct a debugger for the web-based environment which could help students to understand how program runs. In this section, we describe the unique features of the notional machine we have developed: bi-directionality and a two level hierarchy

Bi-directionality

A traditional IDE debugger requires users to set up breaking points. The system will encounter a break point and suspend the execution where the break point is. The user then can step forward to reveal debug information. However, if users are step over the break point, they must excute the steps again until they reach the break point.

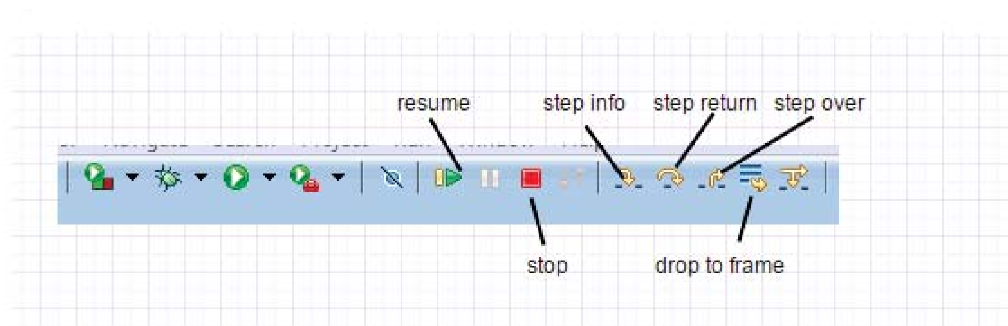

Figure 11: Eclipse Debugger

Figure 11 shows how Eclipse presents a directional debugger.
A bi-directionality provides an ability to reverse stepping . This can potentially save a lot of time and effort, especially when there are deep and nest program

structures. In our notional machine, we support reverse stepping through the program code but also provide a function to stepping through game states. We believe this feature is intuitive for new students as they are used to the reverse function of media players.

Two level hierarchy

The game board shows the high-level state of the program, while the code view can provide in-sights how this state is computed. This type of notional machine can help to find out what the behaviour of the program is. Figure 12 shows an example. It demonstrates the algorithm for picking the largest prime number. The assignment indicates that the value 17 is assigned to the variable `largest` at this state.

Figure 12: Program code Comprehension: Assignment

The Figure 13 shows another example. The next execution is a method invocation which is `isPrime()` method.

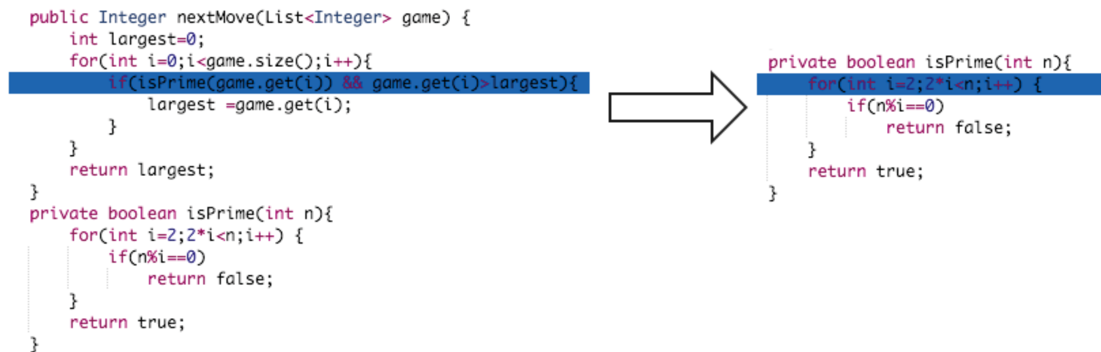

Figure 13: Program code Comprehension: Method Call

The board game's state is changing throughout the game play. To enhance the semantic interpretation, we apply the conceptual model theory (Section 2.3.2). As Figure 14 demonstrates, the game state changes from the player red's turn to the player blue's turn. It tells us the player blue picks the number 3 at this state. We represent this transition in the form of 10*10 game board which consists 100 numbers.

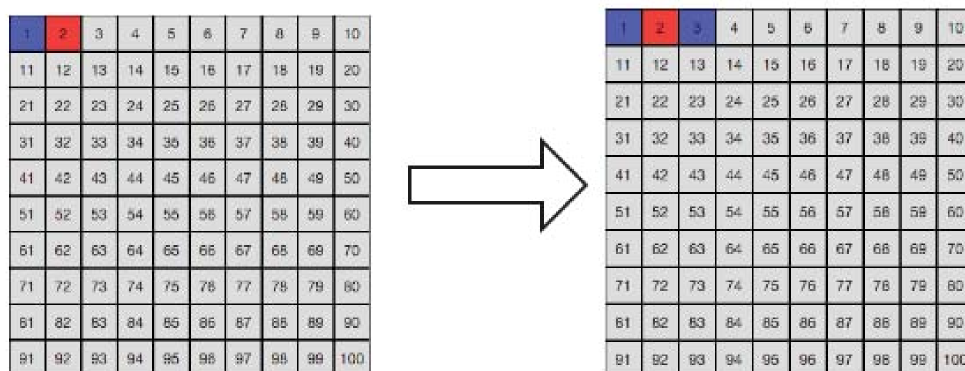

Figure 14: Game strategy comprehension: State Changing

One disadvantage of using only the visual representation is that it only demon-

strates the state changes on the game board. It does not reveal how the next game state is computed (eg: how the variables change). The Figure 15 shows the textual view which can be used to explain what the largest prime number is. At this state, we are investigating the blue bot's third move and the program state at line 25. Four variables are displayed with its value. With the textual representation, the layered notional machine not only reveals the semantics of program code in a form of text representation, but also can represent the current game state visually.

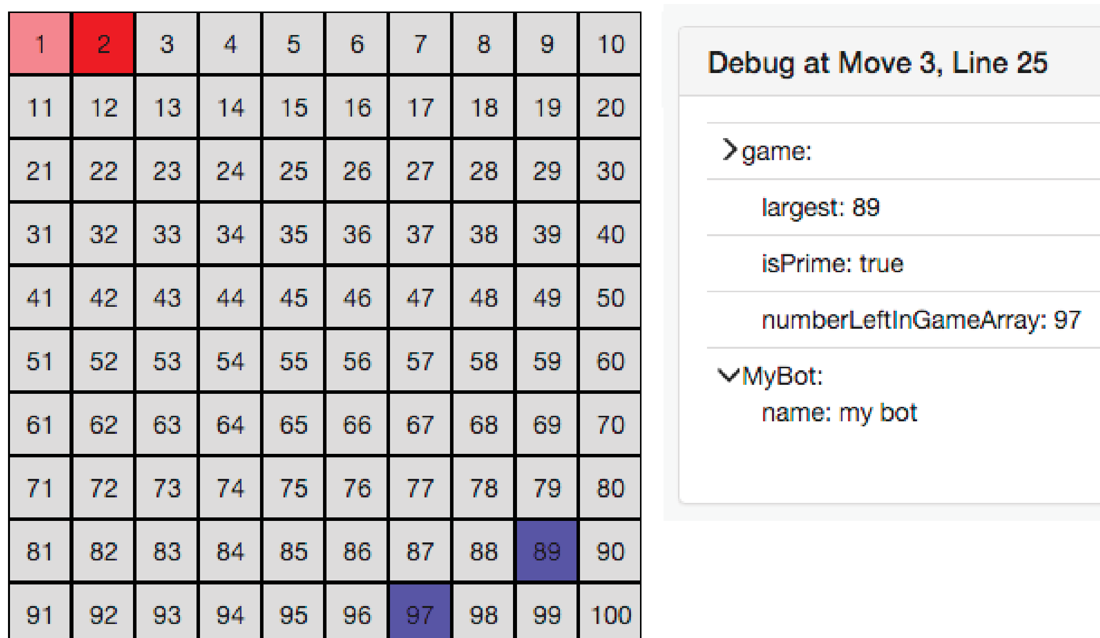

Figure 15: Visual plus texture view

Design and Implementation

This section describes the design and implementation of traceability on both client and server side.

Client

In this section, we present the design of a new user interface for traceability to facilitate the reverse stepping and two layers notional machine. The new user interface is based on the SoGaCo platform [18]. The following list is a summary of the various parts of the user interfaces.

1. Figure 16 shows the test page in the existing SoGaCo user interface. For the purpose of usability, we decided to redesign the user interface. The game board and editor are displayed within the same page (Figure 17). This facilitates the presentation of the notional machine, where both views are necessary in order to display programming states.

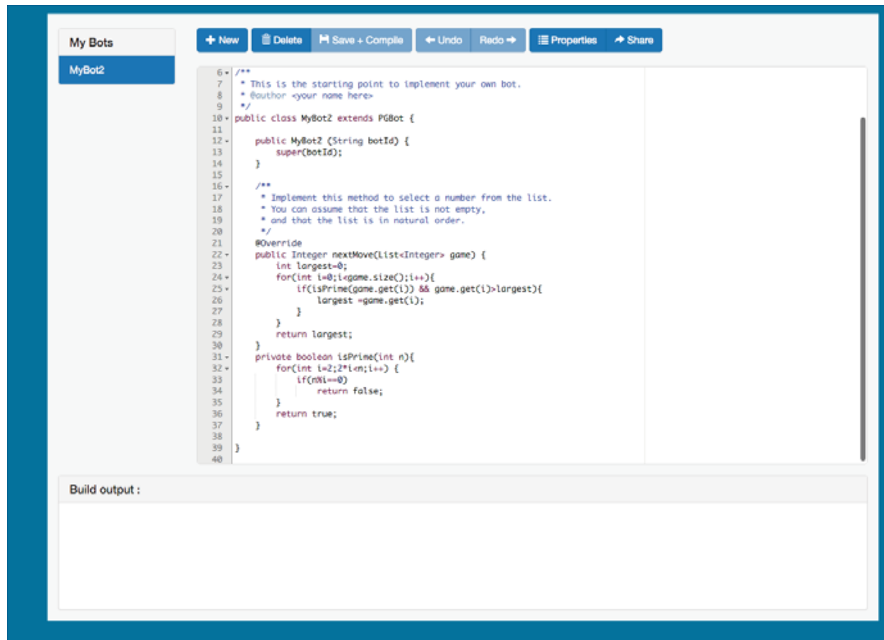

Figure 16: Editor page

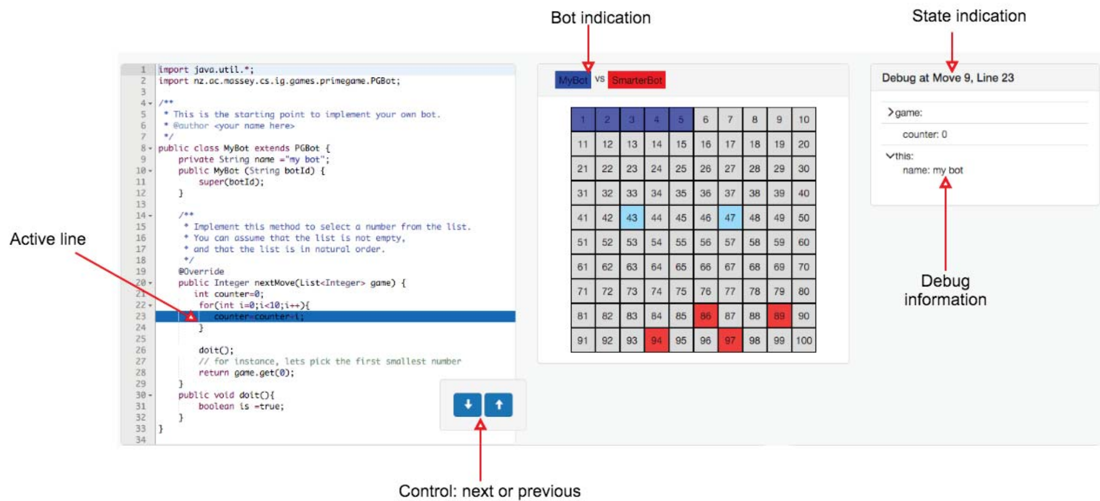

Figure 17: Testing page

2. Users can toggle the debug view. The debug view, the game board and the editor are positioned within the same page so that users can follow the

changing states on the game board and the debug pane simultaneously. It is also designed as a collapsible block along with other blocks. The debugging pane in debug view are also collapsible. When there is more demand for space, users can close it.

3. Bot selection. There is a new way to select the bots to play with. A togglable sidebar contains user's bot and a drop-down list includes other bots to play with. The source code of user's bot is displayed in a editor (Figure 18).

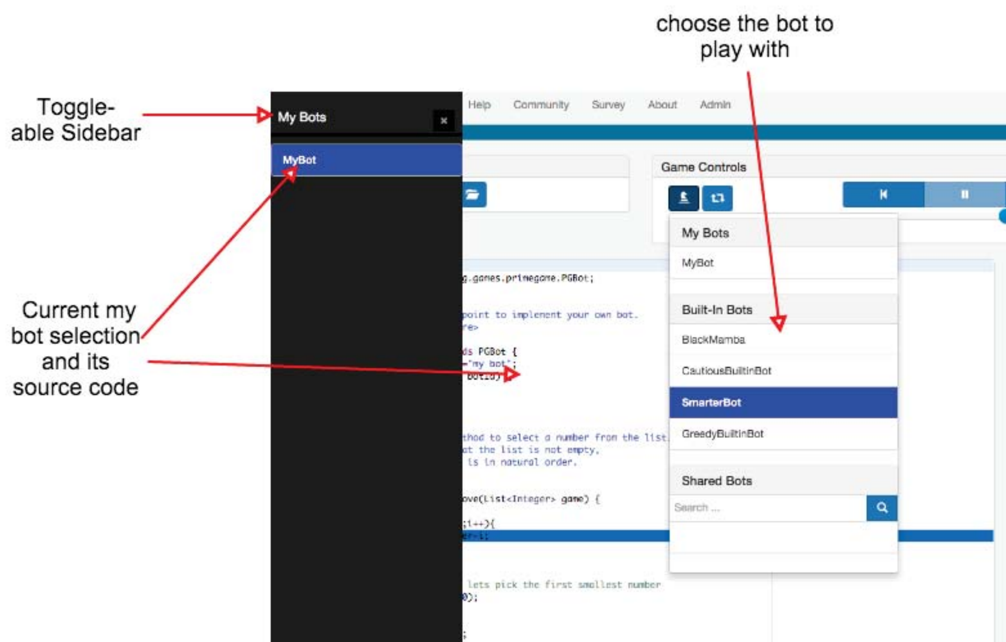

Figure 18: Bot Selection

After both bots have been selected, users can click the play button to bring up the dialog shown in Figure 19. Users must choose which bot plays the first move. This is very important in the turn-based game because the order can affect the game outcome. Users are also able to choose whether to run the game in debug mode.

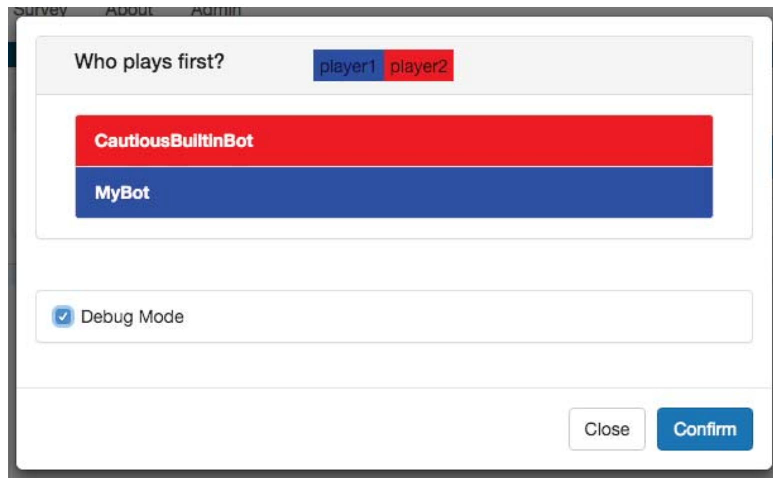

Figure 19: Choose Who Plays First

4. Bi-directional debug control. A debugging control button allows to jump to the next or previous line and display the corresponding tracing information. It is very convenient to control because it is floating above the editor. If the contents in the editor overflows, the control still sticks to the right bottom corner (Figure 17).

Server

In this section, we present the implementation of traceability on the server side. It includes two instrumentation methods: source code instrumentation and byte code instrumentation. We also discuss five different methods to encode and compress the results from tracing in order to minimise the server work-load.

Snapshots

The tracing mechanism is based on the notion of snapshots. A snapshot represents the current state of variables in a running program. Both source code instrumentation and byte code instrumentation can be used to extract snapshots with a

structure as shown in Figure 20.

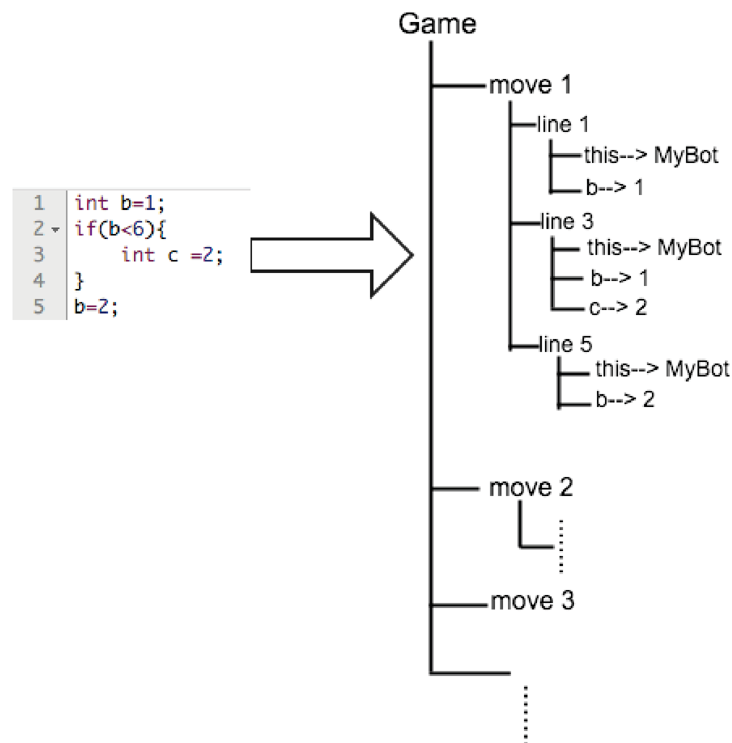

Figure 20: Snapshot

The snapshots use the key-value set to map variables on the stack to values. We create a snapshot whenever we encounter that (1) variables are added to or removed from the current scope (2) variable values have changed. For example, Figure 20 shows a code snippet within a move. After `b=1` has been executed, we take a snapshot of integer `b` and the current object identified by the **`this`** reference. Then a condition expression is encountered and new snapshot is taken that includes `c` and `b`. After the condition expression, the value of variable `b` changes and the new value is captured. But the variable `c` needs to be dropped. This is because the variable `c` is no longer exists within the scope as the frame drops when the execution exits the block.

Implementing tracing using source code instrumentation

The library we choose is SUN Compiler API as the library has a AST like visitor. The process of source code instrumentation is straightforward as the Figure 21 shows below. There are two main actions: AST analysis and string manipulation.

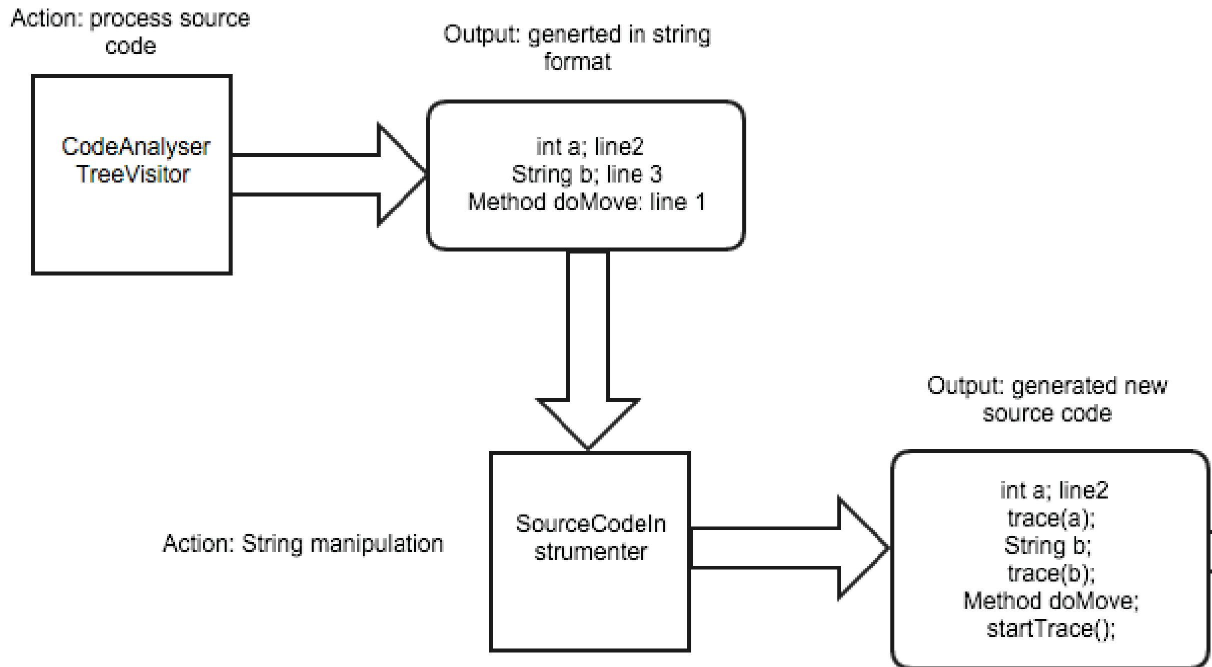

Figure 21: Method Flow of Source Code Instrumentation

- AST analysis: We use the visitor pattern [24] to travel through code. Next, we retrieve useful information like line numbers, variables, methods and class names. They are stored as key-value set. Figure 22) shows the simplified AST structure of the prime number strategy (Listing 4). The MyBot class has two methods: `nextMove()` and `isPrime()`. In the `nextMove()` method, there are two variables that align with the corresponding lines. The `isPrime()` method only has one variable which is located at line 3.

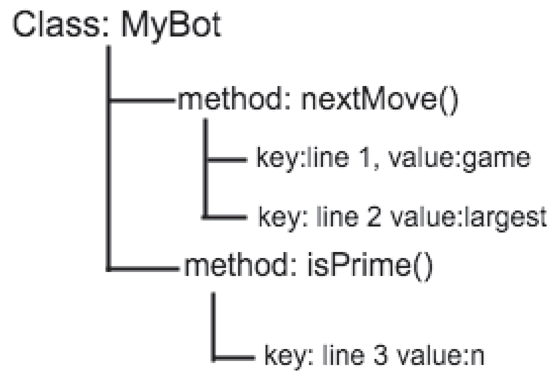

Figure 22: Nested Map data structure for Source Code Instrumentation

- String manipulation. The AST analysis phase results in a nested map (Figure 22). In this phase, we take source code as `inputstream` and read it line by line. Whenever the line number is matching, we register a tracing event by putting a callback statement after that line. For example, when we parse line 3 which has variable `n` as a key, we inject the callback statement "`tracing(n,3)`" to enable the tracing event. To prevent new registration statements change the original source code lines, we use a temporary buffer to rewrite the whole source code. At the end, instrumented code is generated and ready for compilation.

Implementing tracing using byte code instrumentation

The byte code instrumentation library we choose is ASM 5.0. It provides an AST-like structure to manipulate the byte code. The byte code instrumentation consists of two parts (Figure 23): a class visitor and a register. The class visitor is responsible for retrieving variable table and finding out what variable scope is before register tracing events. The register is responsible for injecting tracing events based on the results from the class visitor. Both the class visitor and register

are using the same ASM `MethodVisitor` to extract the information from the byte code. The reason that we use a two pass strategy is because the variable table is visited after visiting set of instructions in every `MethodVisitor`.

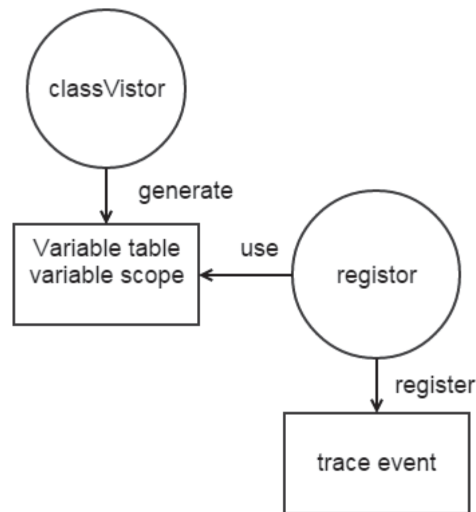

Figure 23: Byte code Instrumentation

An example of local variable table is shown in following Table 2:

Table 2: Local Variable Table

Name	Signature	Start	End	Index
number	I	0	2	1
string	Ljava/lang/String;	0	1	2
this	LmyBot/bot;	0	2	0

The signature indicates the type of a local variable. It could be either a primitive type or an object. For example, `I` indicates the values are 32-bit signed two's-complement integers and `LClassName` is an instance of class [36]. The Start is the first instruction corresponding to the scope of a local variable (inclusive) and the End is the last instruction corresponding to the scope of a local variable

(exclusive). The Start and End indicate where to add and drop variables as the frame changes. The index shows the index of local variable on the stack.

Each of snapshot is taken at where the store and invoke instructions are. There is no need to capture load instruction because we only track the final state of variables. There are various store instructions.

- ASTORE: store a reference into a local variable
- ISTORE/DSTORE/FSTORE/LSTORE: store a integer/double/float/long value into a local variable.
- AASTORE: store a reference in a array.
- IASTORE/DASTORE/FASTORE/LASTORE: store a integer/double/float/long value in a array.

Besides the store instructions, we also trace method invocations.

- INVOKEINTERFACE:invokes an interface method on object and puts the result on the stack
- INVOKEVIRTUAL:invoke virtual method on object and puts the result on the stack
- INVOKESPECIAL:invoke instance, private method, and instance initialization method invocations

`INVOKESTATIC` and `INVOKEDYNAMIC` are other two invokes in Java byte code. The SoGaCo verifier rejects bots with static fields and methods due to static members keep the state of multiple instances of the same bot. We do not encounter static methods at this stage. Furthermore, we do not support `INVOKEDYNAMIC`, and

therefore tracing might miss some states generated by dynamic invocation. This will only have an impact on the precision of instrumentation if the supported language level is Java 1.8, and lambda expressions are used in the source code of bots.

To visit a zero operand instruction, we need to trace RETURN, ARETRUN, IRETURN, FRETURN, DRETURN and LRETURN instructions which return a void, a reference or a primitive type.

For field access, we trace `GETFIELD` and `PUTFIELD` which get/set field to a value in an object, where the field is identified by a field reference index in the constant pool. Static field access (`GETSTATIC`,`PUTSTATIC`) can be ignored as the SoGaCo verifier will reject bots using such instructions.

Discussion

We also spot an unexpected interesting behavior in the byte code instrumentation by using ASM 5.0. One of our JUnit tests is to find out how many trace events being registered. There should be 27 trace events but it is less than 27 trace events on a different computer. We investigate it and found out the reason behind it is the JDK version.

```
int last=game.get(game.size()-1)-1;
if(last>4){
    double d=1.1;
    boolean b=true;
};
float f=1;
```

Figure 24: Example code: Miss capture on different JDK version

The JDK running on one PC is 1.8.0_31. The instrumentation picks up the

statement `boolean b=true` and registers a trace event (Figure 24). However, the JDK with version 1.8.0_60 does not capture the boolean variable. The structure of the byte code depends on the version of compiler being used. It makes sense that the boolean variable is never used and the compiler may ignore it in order to optimise byte code. But according to ASM specification, there is no information related to this update about that optimization which causes a neglect of unused variables. More details are provided in Appendix 7.3.

Compression methods and encoding schemes

In this section, we investigate how the state is represented by the snapshots which is extracted though instrumentation. We propose five ways to encode and compress the snapshots.

The traced data is encoded in the JSON format. During the encoding process, we find that the size of data increases substantially with the complexity of the bot. To deal with such a large amount of data and prevent out of memory errors, we propose five methods to encode traced data. They are: baseline encoder, custom encoder, dictionary index method, edit distance method and tree edit distance. Each method has its own advantages, but reducing the total amount of processing time and memory usage are our primary concerns.

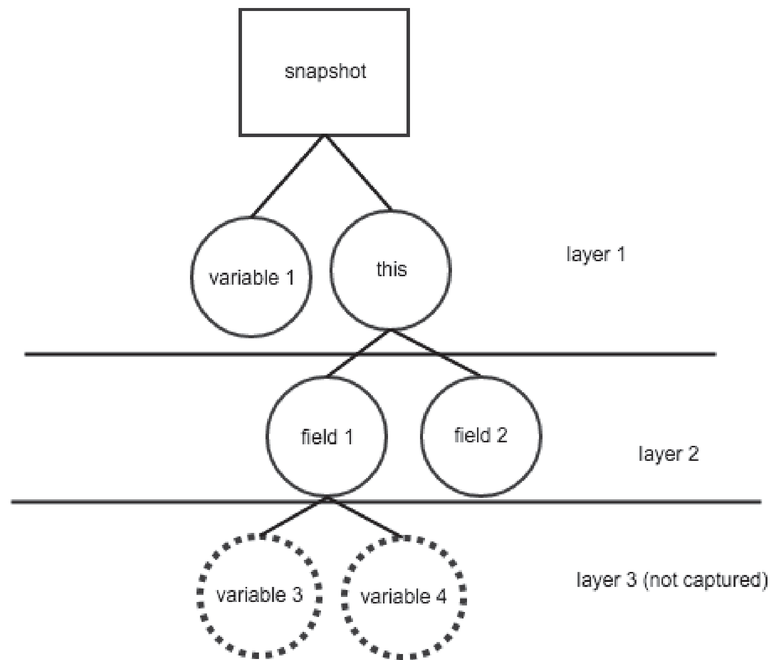

Figure 25: Level of Access: depth 2

Each memory snapshot is created via instrumentation. When a reference variable is encountered, a depth detection is applied (Figure 25). Setting a maximum depth also deals with circular references in the object graph. If the maximum depth reached, then there will be no further encoding process. Take the `LinkedList` for instance, the structure of it is shown as follow (Figure 26). To flatten this structure, we iterate through it and extract the value of each entry. No previous and next entry are captured at this stage.

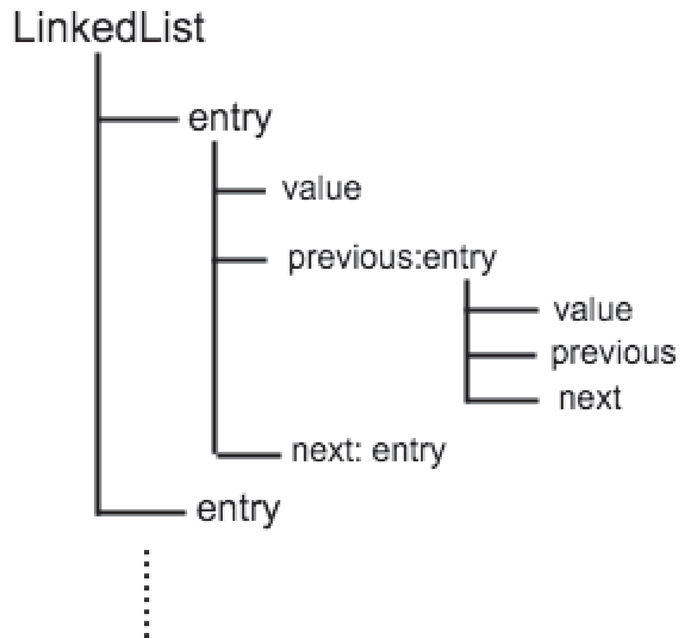

Figure 26: LinkedList structure

The baseLine encoder

The execution process decides how data is encoded. For example, a list of integers from 0 to 10 is encoded as [0,1,2,3,4,5,6,7,8,9]. The baseline encoder basically encodes everything and represents it in the JSON format. The structure is shown in Figure 27.

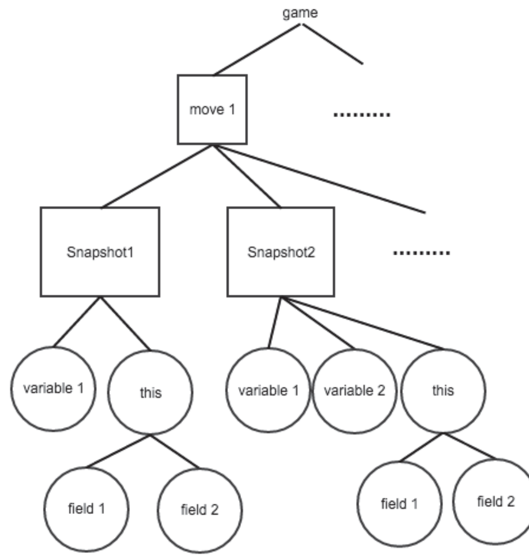

Figure 27: Baseline Encoder Structure

However, a large amount data could be generated due to numbers of states or turns particularly in the board game. For example, The BlackMamba has 217 lines of code and needs 49 moves to complete the game against itself. As a consequence, encoding takes 6,361 million seconds and produces 145 MB of data. This is a heavy burden for the server. On the other hand, baseline encoder does a good job on providing a clear and logic structure for interpreting running program. It also can be easily decoded on the client side without any further computations. But it is an expensive operation in term of memory usage and time consumption.

The custom encoder

The custom encoder encodes everything and shares the same structure with the baseline encoder. However, the custom encoder compresses number arrays. It reduces the number of elements being encoded. Therefore, the total size of data is reduced.

For example, a consecutive number array [0,1,2,3,4,5,6,7,8,9] now becomes a string ["0-10"]. For the non consecutive number array, we decide to find its consecutive parts and keep the non consecutive parts. (e.g.:[0,1,2,3,6,7,8,9] = ["0", "1-3", "6-8", "9"]).

The dictionary index method

The dictionary index method method focuses on the structure of tracing results. Since the baseline and the custom encoder potentially have a large amount of duplications, the dictionary index method alters the structure of tracing results. It stores the state of variable as a key, the value are the position that indicates where the corresponding variable is and its execution order. As a result, a large amount of space can be saved. For instance, a variable (e.g.: `a=1`) remains the same across all the moves in the PrimeGame, the encoded result in JSON would be shown as follows (Listing 9):

Listing 9: A variable that stays the same across all the moves

```
{"value":"a=1","move":[1,2,3,4,5,6,7,8,9,10],"executionID":[1],"line":[22]}
```

If there is a variable that remains the same in a single move, the encoded result in JSON would be shown as follows (Listing 10):

Listing 10: A variable that stays the same in a single move

```
{"value":"a=1","move":[1],"executionID":[1,2,3,4],"line":[22,23,24,25]}
```

If that variable changes, we record it as follows (Listing 11):

Listing 11: Changing variable

```
{{"value":"a=1","move":[1],"executionID":[1],"line":[22]},
{"value":"a=2","move":[2],"executionID":[1],"line":[22]}}
```

The edit distance method

The edit distance provides a function to investigate the differences between each execution state. We use a series of snapshots to indicate the execution states. Each snapshot contains a number of variables. The idea is to stringify each variable in a snapshot and compare them with the one in previous snapshot. The first snapshot is always the base. An example shown as follows (Listing 12):

Listing 12: Adding a variable using edit distance

```
{{"base":true,"snapshot":{"a":"1"},"move":[1],"line":[22],"action":null},
{"base":false,"snapshot":{"b":"2"},"move":[1],"line":[23],"action":"add"}}
```

We only record the remove and the add actions. The add action indicates a variable in the current snapshot does not exist in the previous one. On the other hand, the remove action shows a variable in the previous snapshot does not exist in the current one.

The tree edit distance method

Similar to the edit distance method, the tree edit distance also provides a function to investigate the differences between execution states. However, the tree edit distance takes this step further and focuses on the branch of the tree. Instead of using the string representation of each variable in a snapshot, the tree distance method applies an abstract tree to represent the structure of variables. By comparing each node of variable with the one in the previous snapshot, we can compute the differences by applying following actions: the add, remove and update. At the client side, we can re-compute whole executions by simulating modification actions starting from a base.

Compared to the edit distance method, the tree edit distance method has an advantage. For example, an array of numbers changes from [1,2,3,4] to [1,2,3], this can be represented through the edit distance method as follows (Listing 13)

Listing 13: Remove a number in an array using the edit distance

```
{{"base":true,"snapshot":{"a":"[1,2,3,4]"},"move":[1],"line":[22],"action":null},
{"base":false,"snapshot":{"a":"[1,2,3,4]"},"move":[1],"line":[22],"action":"remove"},
{"base":false,"snapshot":{"a":"[1,2,3]"},"move":[1],"line":[22],"action":"add"}}
```

The first entry is the base. The second entry indicates "remove" action which deletes the variable `a`. The third entry shows variable `a` with new value being added. On the other hand, the tree edit distance method iterates through the array and find the difference (Listing 14).

Listing 14: Remove a number in an array using the tree edit distance

```
{{"base":true,"snapshot":{"a":[1,2,3,4]},"move":[1],"line":[22],"action":null},
{"base":false,"snapshot":{"a":4},"move":[1],"line":[22],"action":"remove"}}
```

However, there is a potential drawback using the tree edit distance method. It produces extra contents in some particular cases. More details will be discussed later (Figure 31)

Experiment Analysis

In this section, we assess the various encoding and instrumentation methods proposed. The validation does not include end user experiments.

Methodology

These experiments were executed on a MacBook Air with a 1.3 GHz Intel Core i5 CPU, 4 GB RAM, OS X version 10.11.2, using a Java HotSpot 64-Bit Server VM (build 25.65-b01, mixed mode). We set the start-up flag to be `-Xmx 3g` for the maximum memory allocation pool for JVM. We use JMH[12], a micro-benchmarking tool that written in Java. It is targeting JVM for building, running and analysing benchmarks. We are using this tool for detailed analysis such as average, variance, standard deviation confidence intervals.

Build performance measurements

The aim of the build performance tests is to find out the effectiveness of two instrumentation methods and to observe the impact on system comparing with the normal build process. The test consists 100 iterations and each iteration has 10 repeated executions. There are several measurement criteria need to be considered:

1. The amount of time to build a bot from compilation to test. We also take measurement of the min/max with standard deviation and confidence interval. Each test contain a warm up stage which has 20 iterations, and each iteration runs 10 times. After warming up, the test consists 100 iterations, and each iteration runs 10 times.

[12]http://openjdk.java.net/projects/code-tools/jmh/

2. The memory usage to complete the build process. We use the `java.lang.Runtime` class to get the free memory and the total memory used by the JVM. Then the used memory is defined as "`total memory-free memory`" which indicates how much of the current heap the JVM is using.

3. As mentioned above in Section 2.2.3, the complexity of the strategy has a strong impact on the actual workload for the instrumentation. We select three different strategies to assess the impact this has. They are GreedyBot (Listing 3), SmartBot (Listing 5) and BlackMamba (Listing 7.4).

The Java profiling is not alway accurate since the performance of the JVM is not completely deterministic [37]. In particular, the JVM needs warm up time to reach optimal performance. It complies methods that executed over a specified number of times to machine code [37]. It is known as compilation threshold.

Runtime performance measurements

After the build process, the bot is ready to be loaded, instantiated and executed. Then we take a measurement to monitor runtime performance where the bots are playing against themselves. Note that only one bot is instrumented during the build and the game play.

We do not include source code instrumentation as this part of runtime test because the results it produces are similar to the results obtained using byte code instrumentation. We have the following factors to consider during the game play tests.

- The amount of time running a game from the beginning until the win condition achieves. The time unit is millisecond.

- The memory required to compose the encoding results. There are five different encoding methods for us to evaluate. For that purpose, we use the `SizeOf` object size estimation library[13]

- The size of file that must be transmitted to the client. It is measured when encoding process end and all results are stored in a txt file.

- The compressed size of the file that must be transmitted to the client. Most clients (including all major web browsers) accept compressions[14]. It is measured after stored `txt` file been compressed into the ZIP file.

- In order to see whether the recording depth affects the file size or not, we measure sizes with an encoding depths of two and three. The captured results in depth 3 are highlighted in red and depth 2 is highlighted in blue. The Figure 25 shows the layer of snapshots.

Build performance (source code instrumentation)

Table 3 shows the results in time for building the bots using source code instrumentation.

Table 3: Time for Building bots using source code instrumentation (milliseconds)

	with instrumentation				no instrumentation			
	average	min/max	stdev	CI(99.9%)	average	min/max	stdev	CI(99.9%)
GreedyBot	40.26	17.83/214.96	19.42	[30.23-42.28]	40.47	21.27/237.24	16.66	[38-42.94]
SmartBot	42.71	21.63/233.31	20.8	[40.54-44.88]	39.59	24.38/214.6	14.35	[39.47-41.72]
BlackMamba	58.76	26.77/260.57	22.34	[51.0-64.92]	56.54	31.75/206.31	11.36	[54.32-58.76]

There is only a slightly difference in time. The GreedyBot and SmartBot require less time to build than the BlackMamba. Note that the unit is milliseconds,

[13]http://mvnrepository.com/artifact/com.carrotsearch/java-sizeof)
[14]The HTTP protocol supports transparent compression of traffic

In fact, the source code instrumentation highly relies on the number of nodes in the AST which need to be instrumented. In this case, The BlackMamba has 227 lines of code need to be instrumented, the SmartBot has 41 lines of code and the GreedyBot has 21 lines of code. Respectively, the BlackMamba requires 58.76 millisecond in average, the SmartBot requires 42.71 milliseconds and the GreedyBot requires 40.26 milliseconds to build. Nonetheless, the impact on build process still remain at minimum.

The Figure 28 using box-plots shows the memory usage for instrumenting bots during the build.

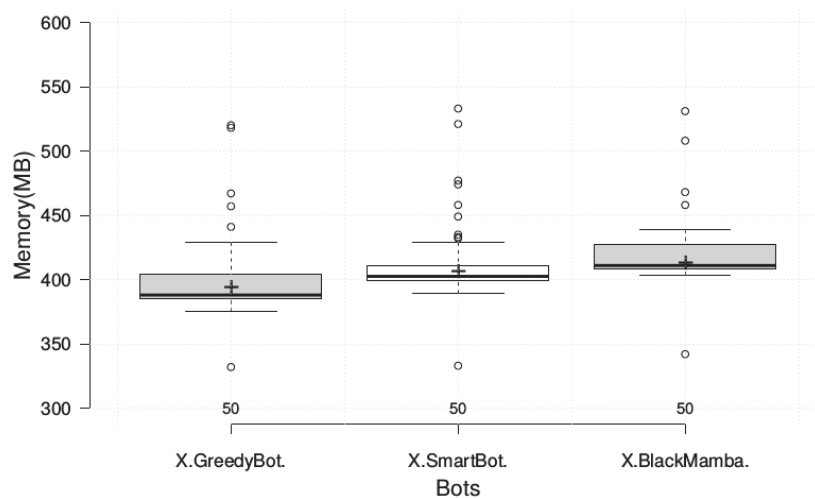

Figure 28: Memory usage: Build Bots with source code instrumentation(MB)

The Figure 29 shows memory usage for building the bots without instrumentation.

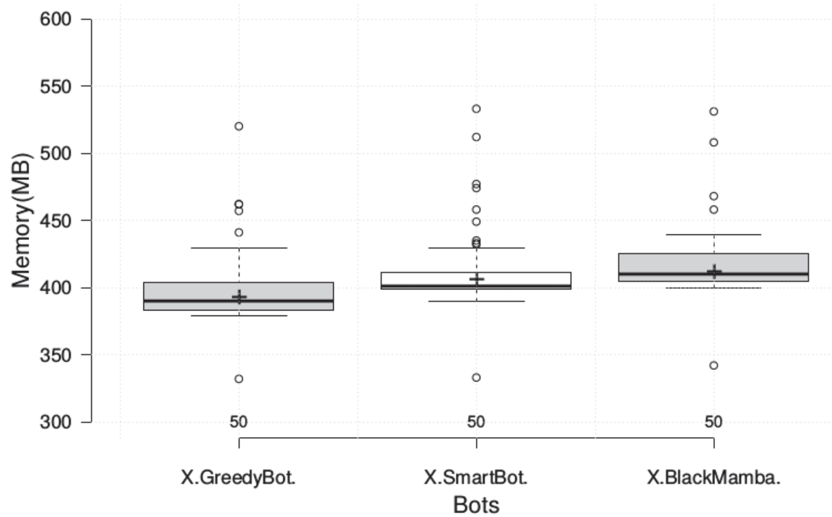

Figure 29: Memory usage: Build Bots without instrumentation(MB)

For the memory usage benchmark, there is no significant difference between building the bot with instrumentation and building the bot without instrumentation. The total memory usage during build process is 400MB approximately. For three different game play strategies, there is no significant differences among them either. Therefore, the source code instrumentation does not have any significant impacts on build process in term of memory consumption.

To summarise, the source code instrumentation does not have a significant impact on the bot build process.

Build performance (byte code instrumentation)

In this section, we conduct a benchmarking on the bot build process using the byte code instrumentation. Table 4 shows the time needed for building the bots using the byte code instrumentation.

Table 4: Time for Building bots using byte code instrumentation(milliseconds)

	with instrumentation				no instrumentation			
	average	min/max	stdev	CI(99.9%)	average	min/max	stdev	CI(99.9%)
GreedyBot	43.67	25.07/240.12	15.56	[41.37-45.98]	40.47	21.27/237.24	16.66	[38-42.94]
SmartBot	49.64	23.86/255.44	21.93	[46.39-52.87]	39.59	24.38/214.6	14.35	[39.47-41.72]
BlackMamba	62.5	35.91/176.16	16.7	[60.03-64.98]	56.54	31.75/206.31	11.36	[54.32-58.76]

The Table 4 shows there is no significant difference in time demanding for building three different bots. The BlackMamba takes slightly longer to build than other bots, and the GreedyBot is the fastest bot to build.

The Figure 30 below shows the memory usage using the byte code instrumentation for building the bots.

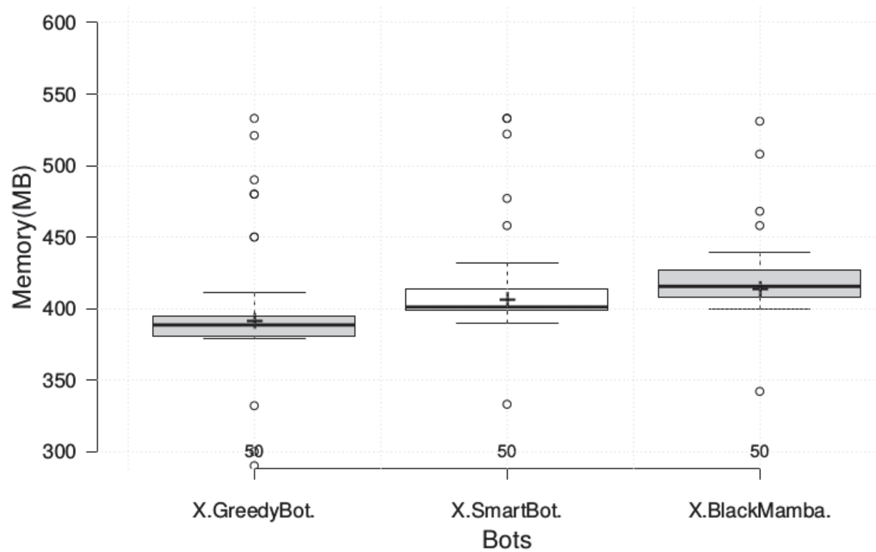

Figure 30: Memory usage: Build Bots Byte code instrumentation(MB)

A complex bot like the BlackMamba takes longer and consumes more memory to build than the other two bots. Comparing building the bot with instrumentation and building the bot without instrumentation (Figure 29) shows no significant

differences.

Runtime performance

In this section, we run performance tests against 5 different encoding schemes discussed in Section 4.2.5. We instantiate the bot class twice and let the two instances play against itself. But only one bot is instrumented.

Table 5: Different Encoding and Compression Result in Time(milliseconds)

		average time	min/max time	stdev	CI(99.9%)
BE[15]	GreedyBot	2.27	1.40/24.14	2.241	[1.51-3.03]
	SmartBot	237	202.85/346.98	27.914	[227.91-246.85]
	BlackMamba	4,521.36	4,266.1/4,950.9	123.58	[4,479.45-4,563.27]
CE[16]	GreedyBot	2	1.06/4.03	0.55	[1.82-2.2]
	SmartBot	185.75	165.19/265.86	15.784	[180.4, 191.1]
	BlackMamba	4,420.62	4,141.79/4,921.09	124.36	[4,378.43-4,462.8]
DI with[17] CE	GreedyBot	2.91	1.36/19.68	1.936	[2.26-3.57]
	SmartBot	274.21	247.61/600.24	39.95	[260.67, 287.76]
	BlackMamba	5,285.56	5,039.92/5,690.12	116.1	[5,246.19-5,324.94]
ED with[18] CE	GreedyBot	2.79	1.24/20.88	2.23	[2.03-3.54]
	SmartBot	238.29	213.51/347.34	24.39	[230.03-246.57]
	BlackMamba	5,226.1	4,937.59/5,720.63	185.4	[5,163-5,288.96]
TED[19]	GreedyBot	3.15	1.13/35.28	3.8	[1.86-4.44]
	SmartBot	610.99	539.31/930.90	76.03	[585.2, 636.77]
	BlackMamba	35,700.11	33,088.76/39,550.22	1,318.25	[35,267.77-39,550.21]

[15] Baseline Encoder
[16] Custom Encoder
[17] Dictionary Index with Custom Encoder
[18] Edit Distance with Custom Encoder
[19] Tree Edit Distance

With the baseline encoder, the GreedyBot requires 2.27 millisecond to complete the game in average. The time consumption of the SmartBot is approximately 104 times more than the GreedyBot requires. We have 99.9% confidence that the average time used by the BlackMamba is between 4479.45 and 4563.27 milliseconds. There is no significant difference between the custom encoder and the baseline encoder for the GreedyBot and the SmartBot time wise. The time benchmark shows the custom encoder and the dictionary index method have similar time consumption except for the BlackMamba. The time consumed by the BlackMamba increases 19.6% from the custom encoder to the dictionary index method.

The time consumption for the tree edit distance is higher than for the other 4 methods. It also has the largest standard deviation. The BlackMamba consumes the most time compared to other methods. It approximately increases 7.07 times.

Table 6 below shows the memory, file size and zip size of tracing results.

Table 6: Different Encoding and Compression Result in Memory(KB). (blue:depth 2, red:depth 3)

		memory (sizeOf)	file size	ZIP size
BE[20]	GreedyBot	13	16	4
		13	16	4
	SmartBot	11,264	11.980.8	119
		11,264	11,980.8	119
	BlackMamba	148,480	155,852.8	1,945.6
		163,840	166,912	2,560
		5	8	4

CE[21]	GreedyBot	5	8	4
	SmartBot	5,120	6,963.2	82
		5,120	6,963.2	82
	BlackMamba	113,664	120,115.2	1,740.8
		129,024	136,192	2,355.2
DI with[22] CE	GreedyBot	3	4	4
		3	4	4
	SmartBot	63	66	4
		63	66	4
	BlackMamba	1,024	2,048	172
		2,048	2,457.6	389
ED with[23] CE	GreedyBot	5	8	4
		5	8	4
	SmartBot	1,024	1,536	82
		1,024	1,536	82
	BlackMamba	70,656	74,649.6	2,251.8
		76,800	81,920	3,072
TED[24]	GreedyBot	11	12	4
		11	12	4
	SmartBot	1,024	1,228.8	86
		1,024	1,228.8	86
	BlackMamba	30,720	32,768	2,150.4
		40,960	45,056	4,096

[20] Baseline Encoder
[21] Custom Encoder
[22] Dictionary Index with Custom Encoder
[23] Edit Distance with Custom Encoder
[24] Tree Edit Distance

In the baseline encoder, the reason that it takes longer is because it captures all tracing information. As a result, the memory usage remains at a very high level. The GreedyBot generates the smallest results with 16 KB and the BlackMamba gives the largest file with 2,560KB (at depth 3). Compared to capturing traces at depth 2 for the BlackMamba, the file size increases 10%. For the GreedyBot and the SmartBot, the file size remains the same due to its plain structure. After the compression, the GreedyBot, SmarBot and BlackMamba produce 4KB, 119KB and 1,945.6KB (at depth 2) compressed files respectively.

The custom encoder uses a pattern to represent consecutive numbers in an array. It shares the same structure as the baseline encoder (Figure 27). For the memory usage, the GreedyBot and SmartBot consume 50% less memory than using the baseline encoder. The BlackMamba consumes 30% less memory than using the baseline encoder. When comparing depth 2 with depth 3 in the BlackMamba using the custom encoder, the file size increases 13% from 1,740.8KB to 2,355.2KB. It has similar growth rate as that of the baseline encoder.

The dictionary index method uses the custom encoder. It does not implement the same snapshots structure as the baseline encoder. In order to reduce the size of the result, a dictionary based structure is proposed. The key is the state of the variable, and the value indicates the lines, moves and execution orders. There is a vast difference between the dictionary index method and the others in term of memory usage and file size. Compared to the custom encoder, the dictionary index method shrinks the size of file from 6,963.2KB to 4KB for the SmartBot and 1,740.8KB to 1,024KB for the BlackMamba. After the compression, the size of file can be reduced to 4KB, 4KB and 172KB for these three bots respectively. Throughout all other four methods, the dictionary index method produces the

minimum file size.

The custom encoder is also applied to the edit distance method. It produces a fairly small file size compared to the baseline encoder. The edit distance method records the number of operations that indicates the transitions between snapshots. The memory usage of the BlackMamba reduces from 148,480KB to 70,656KB compared to the baseline encoder. However, it increases by 69,632KB compared to the dictionary index method. Both of them are using the custom encoder. Both the baseline encoder and the edit distance method use nearly the same amount of time to complete a game.

The tree edit distance employs neither the custom encoder nor the baseline encoder. Its unique structure (Listing 14) determines how variable states are encoded. Unlike the edit distance method, the tree edit distance focuses on the elements that variables consist of, and forms sequence of tree structure which can be explained by a number of operations. As the result, the BlackMamba produces a 32,768KB file and that is nearly half compared to the file produced by the edit distance method. It means there is some redundancy recorded by the edit distance method due to the level of access to a particular variable. The Table 6 shows the file size increases 33% from depth 2 to depth 3.

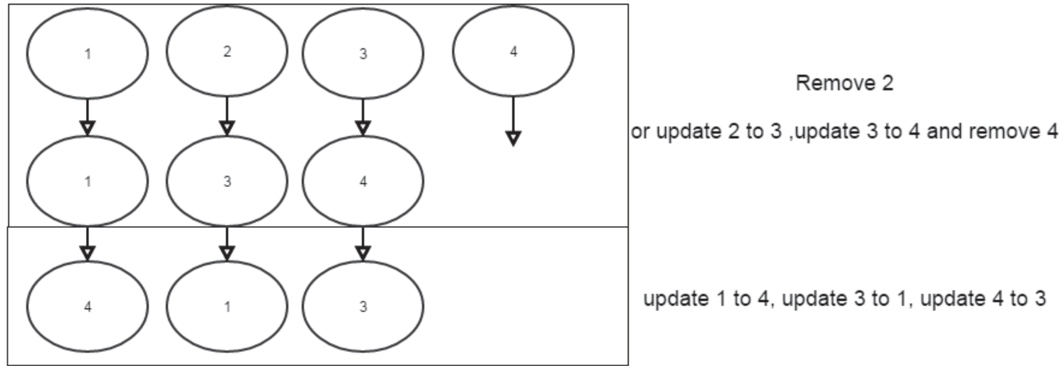

Figure 31: Producing extra contents

Figure 31 shows two cases that could produce extra contents. The first one is the removal of a variable in the middle (assume that the variables have already been ordered). Then the operation can be addressed as “update 2 to 3, update 3 to 4 and remove 4” instead “remove 2” directly. In the seconds case, the order of variables is completely random. A full update operation is performed. In conclusion, a further redundancy check must be applied to prevent the above cases. As a consequence, performance issues arise.

In summary, the edit distance method focuses on a two-level structure: snapshot and variable value. It produces a fairly small amount of data comparing with the custom encoder and baseline encoder. The tree edit distance method focuses on three-level structure: snapshots, variables and their elements. It has an advantage dealing with a complex algorithm like BlackManba. But it takes longer to compute than other four methods. The dictionary index method with a custom encoder has reasonable time consumption and file size. The baseline encoder provides an easy and intuitive way to represent snapshot and its states. However, it produces a huge amount data to be delivered to the client side. In contrast, the

other four methods need more work to decode on the client side.

Conclusion and Future work

Conclusion

In the previous section we have focus on performance evaluation. A number of encoding methods were tested. At the end, we choose the dictionary index method over others because the amount of data reduced. There is no significant difference between each encoding method in time-wise except the tree edit distance method. The three-level structure of the tree edit distance method determine it takes longer to compute than other methods.

The level of game play strategy discussed in section 2.2.3 reveals the fact that the level of difficulty can not only provide more options for students but also can be used as assignments for instructors for assessment. We also find a positive relationship between the level of gaming play strategy and the workload for encoding tracing results. The level of gaming play strategy increases, so dose the demand for memory and time. It does not matter what encoding or comparison methods are applied, the BlackMamba bot always consumes most resources. We also explore the depths that been captured in three bots. The size of encoded file only increases in the BlackMamba because it has an inner class `Move` to represent the game broad.

During bot building, there is no significant difference between the source code and the byte code instrumentation in term of time and memory consumption. Furthermore, the results from Section 5.2 and Section 5.3 indicate the impact that applying instrumentations on the bot build process remain at minimum

Through the instrumentation techniques, we are able to capture the program

states and represent them in a layered notional machine discussed in Section 4.1 and Section 3. The layered notional machine contains a game broad which demonstrate the game strategy in a conceptual level, and the code editor with tracing information provides a comprehension on the semantics of program code for novice learners. We capture the entire executions to facilitate bi-directionality which makes reverse stepping possible.

In conclusion, we have implemented a proof-of-concept notional machine based on a highly scalable web-based platform. We also perform a number of test to verify its feasibility.

Future work

This section describes possible works that may carried by other.

End user validation

This thesis is focused on the proof-of-concept implementation of a layered notional machine. On the conceptual level, we already discuss the potential benefits in gaming which is linked to intrinsic motivation analyses. The transformation from extrinsic to intrinsic motivation involves certain level of satisfaction using our software. The purpose of implementing traceability is to improve programming comprehension for students. Due to the time constraint, this thesis does not include validation for end users. In the future, an end user validation can be conduct to find out if our work translates into better educational outcomes.

This end user study can be conducted through qualitative study and quantitative study. In quantitative study, a question in survey like "how strongly do you agree that the debugger can help me to understand the program?" may provide a

numerical data for satisfaction levels. Also, design some small tasks for users to complete. For example, we could ask users to find a particular variable's value in certain move and measure how long they take to finish it. On the other hand, a qualitative investigation would helps as well (e.g. talking and observing students).

Compression improvement vs Game play strategy

We present 5 compression and encoding method for tracing results. Their performance are all limited to game play strategy. From results discussed in the experiment section, it is clear to see that these 5 compression methods reduce the file size and some of methods do have significant impact on time consumption. For example, the BlackMamba still needs longer to compute than other two bots. We need to find a more efficient method that not only reduce the amount data and time but also balance it with the game play strategy. For example, the custom encoder needs to handle complex data structure. The position data can be represented as `Position(x,y)`. A particular use case is the Tic Tac Toe game. Using a predefined number 1 to represent top center position on game board (`Position(0,1)`). Then, the list of move for a player now become list of numbers, and the custom encoder can easy compress this number array. The tree edit distance method needs an improvement on producing the extra contents (Figure 31) in the future.

Other games

This thesis focused on PrimeGame which is a pure mathematical board game. We would like to see more games to be added to the SoGaCo platform. For instance, The five in row (Gomoku) is a traditional Chinese broad game. It is played on a Go game broad. We would like to see the analysis of the play strategies of Five in row and the traceability of it. Furthermore, we would to know the educational benefits among the different broad games in the context of culture, game complexity and

API complexity.

Appendix

Java Debug Interface Code Example

Listing 15: Monitor field

```
public class FieldMonitor {

  public static final String CLASS_NAME = "Test";
  public static final String FIELD_NAME = "foo";

  public static void main(String[] args)
      throws IOException, InterruptedException {
    // connect
    VirtualMachine vm = new VMAcquirer().connect(8000);

    // set watch field on already loaded classes
    List<ReferenceType> referenceTypes = vm
        .classesByName(CLASS_NAME);
    for (ReferenceType refType : referenceTypes) {
      addFieldWatch(vm, refType);
    }
    // watch for loaded classes
    addClassWatch(vm);

    // resume the vm
    vm.resume();

    // process events
```

```
  EventQueue eventQueue = vm.eventQueue();
  while (true) {
    EventSet eventSet = eventQueue.remove();
    for (Event event : eventSet) {
      if (event instanceof VMDeathEvent
          || event instanceof VMDisconnectEvent) {
        // exit
        return;
      } else if (event instanceof ClassPrepareEvent) {
        // watch field on loaded class
        ClassPrepareEvent classPrepEvent = (ClassPrepareEvent) event;
        ReferenceType refType = classPrepEvent
            .referenceType();
        addFieldWatch(vm, refType);
      } else if (event instanceof ModificationWatchpointEvent) {
        // a Test.foo has changed
        ModificationWatchpointEvent modEvent =
            (ModificationWatchpointEvent) event;
        System.out.println("old="
            + modEvent.valueCurrent());
        System.out.println("new=" + modEvent.valueToBe());
        System.out.println();
      }
    }
    eventSet.resume();
  }
}

/** Watch all classes of name "Test" */
```

```java
  private static void addClassWatch(VirtualMachine vm) {
    EventRequestManager erm = vm.eventRequestManager();
    ClassPrepareRequest classPrepareRequest = erm
        .createClassPrepareRequest();
    classPrepareRequest.addClassFilter(CLASS_NAME);
    classPrepareRequest.setEnabled(true);
  }

  /** Watch field of name "foo" */
  private static void addFieldWatch(VirtualMachine vm,
      ReferenceType refType) {
    EventRequestManager erm = vm.eventRequestManager();
    Field field = refType.fieldByName(FIELD_NAME);
    ModificationWatchpointRequest modificationWatchpointRequest = erm
        .createModificationWatchpointRequest(field);
    modificationWatchpointRequest.setEnabled(true);
  }

}
```

Listing 16: Connect to VM

```java
public class VMAcquirer {

  /**
   * Call this with the localhost port to connect to.
   */
  public VirtualMachine connect(int port)
```

```
    throws IOException {
  String strPort = Integer.toString(port);
  AttachingConnector connector = getConnector();
  try {
    VirtualMachine vm = connect(connector, strPort);
    return vm;
  } catch (IllegalConnectorArgumentsException e) {
    throw new IllegalStateException(e);
  }
}

private AttachingConnector getConnector() {
  VirtualMachineManager vmManager = Bootstrap
      .virtualMachineManager();
  for (Connector connector : vmManager
      .attachingConnectors()) {
    System.out.println(connector.name());
    if ("com.sun.jdi.SocketAttach".equals(connector
        .name())) {
      return (AttachingConnector) connector;
    }
  }
  throw new IllegalStateException();
}

private VirtualMachine connect(
    AttachingConnector connector, String port)
    throws IllegalConnectorArgumentsException,
    IOException {
```

```
    Map<String, Connector.Argument> args = connector
        .defaultArguments();
    Connector.Argument pidArgument = args.get("port");
    if (pidArgument == null) {
      throw new IllegalStateException();
    }
    pidArgument.setValue(port);

    return connector.attach(args);
  }

}
```

Listing 17: Monitored Code

```
public class Test {
  int foo;
  public static void main(String[] args) {
    Random random = new Random();
    Test test = new Test();
    for (int i = 0; i < 10; i++) {
      test.foo = random.nextInt();
      System.out.println(test.foo);
    }
  }

}
```

JavaFlow Code Example

Listing 18: JavaFlow Code Example

```
public class JavaFlowExample {
  public static class MyRunnable implements Runnable {
    public void run() {
      System.out.println("run started!");
      for( int i=0; i < 100; i++ ) {
        echo(i);
      }
    }

    private void echo(int x) {
      System.out.println("echo " + x);
      Continuation.suspend();
    }
  }

  public static void main(String[] args) {
    System.out.println("main started");
    Continuation c = Continuation.startWith(new MyRunnable());
    System.out.println("in main after continuation return");
    while (c != null) {
       c = Continuation.continueWith(c);
       System.out.println("in main");
    }
  }
}
```

ASM byte code

Listing 19: (JDK version: 1.8.0_60)Bytecode Demonstration

```
// class version 52.0 (52)
// access flags 0x21
public class test/ExampleComplexBot extends
   nz/ac/massey/cs/ig/games/primegame/PGBot implements
   nz/ac/massey/cs/ig/core/game/model/instrumentation/InstrumentedBot {

  // compiled from: ExampleComplexBot.java

  // access flags 0x1001
  public synthetic
     Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
     __observer

  // access flags 0x1
  public <init>(Ljava/lang/String;)V
   L0
    LINENUMBER 12 L0
    ALOAD 0
    ALOAD 1
    INVOKESPECIAL nz/ac/massey/cs/ig/games/primegame/PGBot.<init>
        (Ljava/lang/String;)V
   L1
    LINENUMBER 13 L1
    RETURN
   L2
    LOCALVARIABLE this Ltest/ExampleComplexBot; L0 L2 0
```

```
  LOCALVARIABLE botId Ljava/lang/String; L0 L2 1
  MAXSTACK = 2
  MAXLOCALS = 2

// access flags 0x1
// signature
    (Ljava/util/List<Ljava/lang/Integer;>;)Ljava/lang/Integer;
// declaration: java.lang.Integer
    nextMove(java.util.List<java.lang.Integer>)
public nextMove(Ljava/util/List;)Ljava/lang/Integer;
  ALOAD 0
  GETFIELD test/ExampleComplexBot.__observer :
      Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
  INVOKEINTERFACE
      nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.beginMove
      ()V
  ALOAD 0
  GETFIELD test/ExampleComplexBot.__observer :
      Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
  LDC 0
  INVOKEINTERFACE
      nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.createSnapshot
      (I)V
  ALOAD 0
  GETFIELD test/ExampleComplexBot.__observer :
      Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
  LDC "test/ExampleComplexBot"
  ALOAD 0
```

```
 INVOKEINTERFACE
    nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
    (Ljava/lang/String;Ljava/lang/Object;)V
L0
 LINENUMBER 17 L0
 ALOAD 1
 ALOAD 1
 INVOKEINTERFACE java/util/List.size ()I
 ALOAD 0
 GETFIELD test/ExampleComplexBot.__observer :
    Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
 LDC 17
 INVOKEINTERFACE
    nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.createSnapshot
    (I)V
 ALOAD 0
 GETFIELD test/ExampleComplexBot.__observer :
    Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
 LDC "game"
 ALOAD 1
 INVOKEINTERFACE
    nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
    (Ljava/lang/String;Ljava/lang/Object;)V
 ALOAD 0
 GETFIELD test/ExampleComplexBot.__observer :
    Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
 LDC "test/ExampleComplexBot"
 ALOAD 0
```

```
INVOKEINTERFACE
   nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
   (Ljava/lang/String;Ljava/lang/Object;)V
ICONST_1
ISUB
INVOKEINTERFACE java/util/List.get (I)Ljava/lang/Object;
ALOAD 0
GETFIELD test/ExampleComplexBot.__observer :
   Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
LDC 17
INVOKEINTERFACE
   nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.createSnapshot
   (I)V
ALOAD 0
GETFIELD test/ExampleComplexBot.__observer :
   Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
LDC "game"
ALOAD 1
INVOKEINTERFACE
   nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
   (Ljava/lang/String;Ljava/lang/Object;)V
ALOAD 0
GETFIELD test/ExampleComplexBot.__observer :
   Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
LDC "test/ExampleComplexBot"
ALOAD 0
INVOKEINTERFACE
   nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
   (Ljava/lang/String;Ljava/lang/Object;)V
```

```
CHECKCAST java/lang/Integer
INVOKEVIRTUAL java/lang/Integer.intValue ()I
ALOAD 0
GETFIELD test/ExampleComplexBot.__observer :
   Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
LDC 17
INVOKEINTERFACE
   nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.createSnapshot
   (I)V
ALOAD 0
GETFIELD test/ExampleComplexBot.__observer :
   Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
LDC "game"
ALOAD 1
INVOKEINTERFACE
   nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
   (Ljava/lang/String;Ljava/lang/Object;)V
ALOAD 0
GETFIELD test/ExampleComplexBot.__observer :
   Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
LDC "test/ExampleComplexBot"
ALOAD 0
INVOKEINTERFACE
   nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
   (Ljava/lang/String;Ljava/lang/Object;)V
ICONST_1
ISUB
ISTORE 2
ALOAD 0
```

```
GETFIELD test/ExampleComplexBot.__observer :
    Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
LDC 17
INVOKEINTERFACE
    nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.createSnapshot
    (I)V
ALOAD 0
GETFIELD test/ExampleComplexBot.__observer :
    Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
LDC "game"
ALOAD 1
INVOKEINTERFACE
    nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
    (Ljava/lang/String;Ljava/lang/Object;)V
ALOAD 0
GETFIELD test/ExampleComplexBot.__observer :
    Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
LDC "last"
ILOAD 2
INVOKEINTERFACE
    nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
    (Ljava/lang/String;I)V
ALOAD 0
GETFIELD test/ExampleComplexBot.__observer :
    Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
LDC "test/ExampleComplexBot"
ALOAD 0
INVOKEINTERFACE
    nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
```

```
    (Ljava/lang/String;Ljava/lang/Object;)V
L1
 LINENUMBER 18 L1
 ILOAD 2
 ICONST_4
 IF_ICMPLE L2
 LDC 1.1
 DSTORE 3
 ALOAD 0
 GETFIELD test/ExampleComplexBot.__observer :
    Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
 LDC 18
 INVOKEINTERFACE
    nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.createSnapshot
    (I)V
 ALOAD 0
 GETFIELD test/ExampleComplexBot.__observer :
    Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
 LDC "game"
 ALOAD 1
 INVOKEINTERFACE
    nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
    (Ljava/lang/String;Ljava/lang/Object;)V
 ALOAD 0
 GETFIELD test/ExampleComplexBot.__observer :
    Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
 LDC "last"
 ILOAD 2
```

```
INVOKEINTERFACE
   nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
   (Ljava/lang/String;I)V
ALOAD 0
GETFIELD test/ExampleComplexBot.__observer :
   Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
LDC "d"
DLOAD 3
INVOKEINTERFACE
   nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
   (Ljava/lang/String;D)V
ALOAD 0
GETFIELD test/ExampleComplexBot.__observer :
   Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
LDC "test/ExampleComplexBot"
ALOAD 0
INVOKEINTERFACE
   nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
   (Ljava/lang/String;Ljava/lang/Object;)V
L3
ICONST_1
ISTORE 5
ALOAD 0
GETFIELD test/ExampleComplexBot.__observer :
   Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
LDC 18
INVOKEINTERFACE
   nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.createSnapshot
   (I)V
```

```
ALOAD 0
GETFIELD test/ExampleComplexBot.__observer :
   Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
LDC "game"
ALOAD 1
INVOKEINTERFACE
   nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
   (Ljava/lang/String;Ljava/lang/Object;)V
ALOAD 0
GETFIELD test/ExampleComplexBot.__observer :
   Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
LDC "b"
ILOAD 5
INVOKEINTERFACE
   nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
   (Ljava/lang/String;Z)V
ALOAD 0
GETFIELD test/ExampleComplexBot.__observer :
   Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
LDC "last"
ILOAD 2
INVOKEINTERFACE
   nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
   (Ljava/lang/String;I)V
ALOAD 0
GETFIELD test/ExampleComplexBot.__observer :
   Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
LDC "d"
DLOAD 3
```

```
 INVOKEINTERFACE
    nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
    (Ljava/lang/String;D)V
 ALOAD 0
 GETFIELD test/ExampleComplexBot.__observer :
    Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
 LDC "test/ExampleComplexBot"
 ALOAD 0
 INVOKEINTERFACE
    nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
    (Ljava/lang/String;Ljava/lang/Object;)V
L2
 LINENUMBER 19 L2
FRAME APPEND [I]
 FCONST_1
 FSTORE 3
 ALOAD 0
 GETFIELD test/ExampleComplexBot.__observer :
    Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
 LDC 19
 INVOKEINTERFACE
    nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.createSnapshot
    (I)V
 ALOAD 0
 GETFIELD test/ExampleComplexBot.__observer :
    Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
 LDC "game"
 ALOAD 1
```

```
 INVOKEINTERFACE
    nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
    (Ljava/lang/String;Ljava/lang/Object;)V
 ALOAD 0
 GETFIELD test/ExampleComplexBot.__observer :
    Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
 LDC "last"
 ILOAD 2
 INVOKEINTERFACE
    nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
    (Ljava/lang/String;I)V
 ALOAD 0
 GETFIELD test/ExampleComplexBot.__observer :
    Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
 LDC "f"
 FLOAD 3
 INVOKEINTERFACE
    nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
    (Ljava/lang/String;F)V
 ALOAD 0
 GETFIELD test/ExampleComplexBot.__observer :
    Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
 LDC "test/ExampleComplexBot"
 ALOAD 0
 INVOKEINTERFACE
    nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
    (Ljava/lang/String;Ljava/lang/Object;)V
L4
 LINENUMBER 20 L4
```

```
ALOAD 1
ILOAD 2
INVOKEINTERFACE java/util/List.get (I)Ljava/lang/Object;
ALOAD 0
GETFIELD test/ExampleComplexBot.__observer :
   Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
LDC 20
INVOKEINTERFACE
   nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.createSnapshot
   (I)V
ALOAD 0
GETFIELD test/ExampleComplexBot.__observer :
   Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
LDC "game"
ALOAD 1
INVOKEINTERFACE
   nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
   (Ljava/lang/String;Ljava/lang/Object;)V
ALOAD 0
GETFIELD test/ExampleComplexBot.__observer :
   Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
LDC "last"
ILOAD 2
INVOKEINTERFACE
   nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
   (Ljava/lang/String;I)V
ALOAD 0
GETFIELD test/ExampleComplexBot.__observer :
   Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
```

```
 LDC "f"
 FLOAD 3
 INVOKEINTERFACE
    nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
    (Ljava/lang/String;F)V
 ALOAD 0
 GETFIELD test/ExampleComplexBot.__observer :
    Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
 LDC "test/ExampleComplexBot"
 ALOAD 0
 INVOKEINTERFACE
    nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.addTrace
    (Ljava/lang/String;Ljava/lang/Object;)V
 CHECKCAST java/lang/Integer
 ALOAD 0
 GETFIELD test/ExampleComplexBot.__observer :
    Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
 INVOKEINTERFACE
    nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.endMove
    ()V
 ARETURN
L5
 LOCALVARIABLE d D L3 L2 3
 LOCALVARIABLE b Z L2 L2 5
 LOCALVARIABLE this Ltest/ExampleComplexBot; L0 L5 0
 LOCALVARIABLE game Ljava/util/List; L0 L5 1
 // signature Ljava/util/List<Ljava/lang/Integer;>;
 // declaration: java.util.List<java.lang.Integer>
 LOCALVARIABLE last I L1 L5 2
```

```
  LOCALVARIABLE f F L4 L5 3
  MAXSTACK = 3
  MAXLOCALS = 6

// access flags 0x1041
public synthetic bridge nextMove(Ljava/lang/Object;)Ljava/lang/Object;
 L0
  LINENUMBER 9 L0
  ALOAD 0
  ALOAD 1
  CHECKCAST java/util/List
  INVOKEVIRTUAL test/ExampleComplexBot.nextMove
      (Ljava/util/List;)Ljava/lang/Integer;
  ARETURN
 L1
  LOCALVARIABLE this Ltest/ExampleComplexBot; L0 L1 0
  MAXSTACK = 2
  MAXLOCALS = 2

// access flags 0x1001
public synthetic __initialize(Ljava/lang/Object;)V
 L0
  ALOAD 0
  GETFIELD test/ExampleComplexBot.__observer :
      Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
  IFNONNULL L1
  ALOAD 0
  ALOAD 1
```

```
 PUTFIELD test/ExampleComplexBot.__observer :
    Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
 ALOAD 0
 GETFIELD test/ExampleComplexBot.__observer :
    Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
 ALOAD 0
 INVOKEINTERFACE
    nz/ac/massey/cs/ig/languages/java/instrumentation/Observer.setObservable
    (Ljava/lang/Object;)V
 GOTO L2
L1
FRAME SAME
 NEW java/lang/UnsupportedOperationException
 DUP
 INVOKESPECIAL java/lang/UnsupportedOperationException.<init> ()V
 ATHROW
L2
FRAME SAME
 RETURN
L3
 LOCALVARIABLE this Ltest/ExampleComplexBot; L0 L3 0
 LOCALVARIABLE observer
    Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer; L0
    L3 1
 MAXSTACK = 2
 MAXLOCALS = 2

// access flags 0x1
public __getObserver()Ljava/lang/Object;
```

```
  L0
   ALOAD 0
   GETFIELD test/ExampleComplexBot.__observer :
       Lnz/ac/massey/cs/ig/languages/java/instrumentation/Observer;
   ARETURN
  L1
   LOCALVARIABLE this Ltest/ExampleComplexBot; L0 L1 0
   MAXSTACK = 1
   MAXLOCALS = 1
}
```

BlackMamba

Listing 20: BlackMamba Source Code

```
public class BlackMamba extends PGBot {

  public BlackMamba (String botId) {
    super(botId);
  }

  private static final int MIN_NR_OF_MOVES_TO_CONSIDER = 10;

  private int[] convertIntegerSetToArray(Set<Integer> set) {
    int[] array = new int[set.size()];
    int i = 0;
    for (int number : set)
      array[i++] = number;
    return array;
```

```
}

private int sumFactors(int n, int[] numbers) {
   int sum = 0;
   for (int i = 0; i < n; i++) {
      if (numbers[n] % numbers[i] == 0) {
         sum += numbers[i];
      }
   }
   return sum;
}


public class Move implements Comparable<Move> {

   private int move;

   private int gain;

   private Move(int move, int gain) {
      this.move = move;
      this.gain = gain;
   }


   public int compareTo(Move o) {
      return o.gain - this.gain;
   }
```

```
        public String toString() {
            return move + " (" + gain + ")";
        }
    }

    @Override
    public Integer nextMove(List<Integer> game) {
         Set<Integer> availableNumbers = new TreeSet<Integer>();

         availableNumbers.addAll(game);

        // determine the size for the set of best moves to look into..
        // here we take MIN_NR_OF_MOVES_TO_CONSIDER, or 10% of the board
            size,
        // whichever is more
        int maxNrOfMoves = Math.max(MIN_NR_OF_MOVES_TO_CONSIDER, (int)
            (availableNumbers.size() * 0.1));

        // get set of best (locally optimised) moves
        TreeSet<Move> topMoves = getTopMoves(availableNumbers,
            maxNrOfMoves);

        // select the best overall move
        Integer move = selectMove(topMoves, availableNumbers,
            maxNrOfMoves);

        // return the move
        return move;
```

```
}

private Integer selectMove(TreeSet<Move> moves, Set<Integer>
    availableNumbers, int maxNrOfLevel2Moves) {
   Iterator<Move> it = moves.iterator();
   int level1MovesToEvaluate = 10;
   int bestScore = Integer.MIN_VALUE;
   Move bestMove = moves.iterator().next();
   for (int i = 0; it.hasNext() && i < level1MovesToEvaluate; i++) {
      Move m = it.next();
      Set<Integer> numbersLeft =
          computeFollowingBoard(availableNumbers, m.move);
      TreeSet<Move> counterMoves = getTopMoves(numbersLeft,
          maxNrOfLevel2Moves);
      // calculate the score over both levels (move and possible
          counter
      // move)
      if (counterMoves.size() > 0) {
         int level2Score = m.gain -
             counterMoves.iterator().next().gain;
         if (level2Score > bestScore) {
            bestMove = m;
            bestScore = level2Score;
         }
      }
   }
   return bestMove.move;
}
```

```java
private Set<Integer> computeFollowingBoard(Set<Integer> currentBoard,
     int move) {
    Set<Integer> newNumbers = new TreeSet<Integer>(currentBoard);
    newNumbers.remove(move);
    for (Iterator<Integer> it = newNumbers.iterator(); it.hasNext();) {
        int n = it.next();
        if (n > Math.ceil(move / 2)) {
            break;
        }
        if (move % n == 0) {
            it.remove();
        }
    }
    return newNumbers;
}


private TreeSet<Move> getTopMoves(Set<Integer> availableNumbers, int
     maxNrOfMoves) {
    TreeSet<Move> topMoves = new TreeSet<Move>();
    int[] numbers = convertIntegerSetToArray(availableNumbers);
    for (int i = numbers.length - 1; i >= 0; i--) {
        int localScore = numbers[i] - sumFactors(i, numbers);
        Move move = new Move(numbers[i], localScore);
        // add the move to the queue (which remains sorted)
        topMoves.add(move);
        // only keep maxNrOfMoves in queue
        if (topMoves.size() > maxNrOfMoves) {
            topMoves.remove(topMoves.last());
```

```
            }

            if (numbers[i] <= topMoves.last().gain) {
               break;
            }

         }
         return topMoves;
      }
   }
```

Java Parser

Listing 21: JavaParser Code Example

```
    private static CompilationUnit createCU() {
        CompilationUnit cu = new CompilationUnit();
        // set the package
        cu.setPackage(new
            PackageDeclaration(ASTHelper.createNameExpr("java.parser.test")));

        // create the type declaration
        ClassOrInterfaceDeclaration type = new
            ClassOrInterfaceDeclaration(ModifierSet.PUBLIC, false,
            "GeneratedClass");
        ASTHelper.addTypeDeclaration(cu, type);

        // create a method
```

```
    MethodDeclaration method = new
        MethodDeclaration(ModifierSet.PUBLIC, ASTHelper.VOID_TYPE,
        "main");
    method.setModifiers(ModifierSet.addModifier(method.getModifiers(),
        ModifierSet.STATIC));
    ASTHelper.addMember(type, method);

    // add a parameter to the method
    Parameter param =
        ASTHelper.createParameter(ASTHelper.createReferenceType("String",
        0), "args");
    param.setVarArgs(true);
    ASTHelper.addParameter(method, param);

    // add a body to the method
    BlockStmt block = new BlockStmt();
    method.setBody(block);

    // add a statement do the method body
    NameExpr clazz = new NameExpr("System");
    FieldAccessExpr field = new FieldAccessExpr(clazz, "out");
    MethodCallExpr call = new MethodCallExpr(field, "println");
    ASTHelper.addArgument(call, new StringLiteralExpr("Hello
        World!"));
    ASTHelper.addStmt(block, call);

    return cu;
}
```

References

[1] The byte code engineering library website. https://commons.apache.org/proper/commons-bcel/index.html. Accessed: 2016-02-28.

[2] Java programming language agents api. https://docs.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html. Accessed: 2016-02-29.

[3] Jep 220: Modular run-time images. http://openjdk.java.net/jeps/220. Accessed: 2016-02-16.

[4] Why does clover use source code instrumentation. https://confluence.atlassian.com/pages/viewpage.action?pageId=79986998. Accessed: 2016-02-15.

[5] M. Al-Bow, D. Austin, J. Edgington, R. Fajardo, J. Fishburn, C. Lara, S. Leutenegger, and S. Meyer. Using greenfoot and games to teach rising 9th and 10th grade novice programmers. In *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, pages 55–59. ACM, 2008.

[6] D. J. Angel, J. R. Kumorek, F. Morshed, and D. A. Seidel. Byte code instrumentation, Nov. 6 2001. US Patent 6,314,558.

[7] B. Apache. Byte code engineering library, november 2009, 2009.

[8] N. A. Bartolome, A. M. Zorrilla, and B. G. Zapirain. Can game-based therapies be trusted? is game-based education effective? a systematic review of the serious games for health and education. In *Computer Games (CGAMES), 2011 16th International Conference on*, pages 275–282. IEEE, 2011.

[9] J. Bennedsen and M. E. Caspersen. Failure rates in introductory programming. *ACM SIGCSE Bulletin*, 39(2):32–36, 2007.

[10] B. Bergeron. Developing serious games (game development series). 2006.

[11] R. Blunt. Does game-based learning work? results from three recent studies. In *Proceedings of the Interservice/Industry Training, Simulation, & Education Conference*, pages 945–955, 2007.

[12] T. E. Bosch. Using online social networking for teaching and learning: Facebook use at the university of cape town. *Communicatio: South African Journal for Communication Theory and Research*, 35(2):185–200, 2009.

[13] B. D. Boulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1):57–73, 1986.

[14] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30:19, 2002.

[15] J. C. Burguillo. Using game theory and competition-based learning to stimulate student motivation and performance. *Computers & Education*, 55(2):566–575, 2010.

[16] S. Chiba. Load-time structural reflection in java. In *ECOOP 2000—Object-Oriented Programming*, pages 313–336. Springer, 2000.

[17] A. Commons. Javaflow, 2009. *URL http://commons. apache. org/sandbox/-javaflow.*

[18] J. Dietrich, J. Tandler, L. Sui, and M. Meyer. The primegame revolutions: A cloud-based collaborative environment for teaching introductory program-

ming. In *Proceedings of the ASWEC 2015 24th Australasian Software Engineering Conference*, pages 8–12. ACM, 2015.

[19] P. Drake and K. Sung. Teaching introductory programming with popular board games. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 619–624. ACM, 2011.

[20] S. Ducasse, A. Lienhard, and L. Renggli. Seaside: A flexible environment for building dynamic web applications. *Software, IEEE*, 24(5):56–63, 2007.

[21] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol–http/1.1, 1999.

[22] M. Ford and S. Venema. Assessing the success of an introductory programming course. *Journal of Information Technology Education: Research*, 9(1):133–145, 2010.

[23] N. Fraser. Blockly website, 2014.

[24] E. Gamma. *Design patterns: elements of reusable object-oriented software.* Pearson Education India, 1995.

[25] J. Gosling. *The Java language specification.* Addison-Wesley Professional, 2000.

[26] K. Hartness. Robocode: using games to teach artificial intelligence. *Journal of Computing Sciences in Colleges*, 19(4):287–291, 2004.

[27] A. Hensman. Evaluation of robocode as a teaching tool for computer programming. 2007.

[28] J. Howells. Software as a service (saas). *Wiley Encyclopedia of Management*, 2014.

[29] T. Java. Platform debugger architecture, 1999.

[30] M. Klawe. The educational potential of electronic games and the e-gems project. In *Proceedings of the ED-MEDIA 94 World Conference on Educational Multimedia and Hypermedia. Panel discussion 'Can electronic games make a positive contribution to the learning of mathematics and science in the intermediate classroom*, 1994.

[31] M. Kölling. The greenfoot programming environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):14, 2010.

[32] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen. A study of the difficulties of novice programmers. In *ACM SIGCSE Bulletin*, volume 37, pages 14–18. ACM, 2005.

[33] K. M. Law, V. C. Lee, and Y.-T. Yu. Learning motivation in e-learning facilitated computer programming courses. *Computers & Education*, 55(1):218–228, 2010.

[34] M. K. Lee, C. M. Cheung, and Z. Chen. Acceptance of internet-based learning medium: the role of extrinsic and intrinsic motivation. *Information & management*, 42(8):1095–1104, 2005.

[35] R. B.-B. Levy and M. Ben-Ari. We work so hard and they don't use it: acceptance of software tools by teachers. *ACM SIGCSE Bulletin*, 39(3):246–250, 2007.

[36] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[37] E. Machkasova, K. Arhelger, and F. Trinciante. The observer effect of profiling on dynamic java optimizations. In *Proceedings of the 24th ACM SIGPLAN*

conference companion on Object oriented programming systems languages and applications, pages 757–758. ACM, 2009.

[38] A. McGettrick and Y. Timanovsky. Digest of acm educational activities. *ACM Inroads*, 5(1):6–10, 2014.

[39] M. Meyer and J. Fendler. The primegame: Combining skills in undergraduate computer science programmes. *INTED2010 Proceedings*, pages 5454–5465, 2010.

[40] I. Milne and G. Rowe. Difficulties in learning and teaching programming—views of students and tutors. *Education and Information technologies*, 7(1):55–66, 2002.

[41] T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, et al. Exploring the role of visualization and engagement in computer science education. In *ACM SIGCSE Bulletin*, volume 35, pages 131–152. ACM, 2002.

[42] E. Nuutila, S. Törmä, and L. Malmi. Pbl and computer programming—the seven steps method with adaptations. *Computer Science Education*, 15(2):123–142, 2005.

[43] J. O'Kelly and J. P. Gibson. Robocode & problem-based learning: a non-prescriptive approach to teaching programming. *ACM SIGCSE Bulletin*, 38(3):217–221, 2006.

[44] M. Papastergiou. Digital game-based learning in high school computer science education: Impact on educational effectiveness and student motivation. *Computers & Education*, 52(1):1–12, 2009.

[45] B. Prabhakar, C. R. Litecky, and K. Arnett. It skills in a tough job market. *Communications of the ACM*, 48(10):91–94, 2005.

[46] M. Prensky and M. Prensky. *Digital game-based learning*, volume 1. Paragon house St. Paul, MN, 2007.

[47] R. Rajaravivarma. A games-based approach for teaching the introductory programming course. *ACM SIGCSE Bulletin*, 37(4):98–102, 2005.

[48] F. Rennie and T. M. Morrison. *E-learning and social networking handbook: Resources for higher education*. Routledge, 2013.

[49] J. C. Reynolds. The discoveries of continuations. *Lisp and symbolic computation*, 6(3-4):233–247, 1993.

[50] K. E. Ricci. The use of computer-based videogames in knowledge acquisition and retention. *Journal of Interactive Instruction Development*, 7(1):17–22, 1994.

[51] A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172, 2003.

[52] J. Ryoo, F. Fonseca, and D. S. Janzen. Teaching object-oriented software engineering through problem-based learning in the context of game design. In *Software Engineering Education and Training, 2008. CSEET'08. IEEE 21st Conference on*, pages 137–144. IEEE, 2008.

[53] J. Sorva. Notional machines and introductory programming education. *ACM Transactions on Computing Education (TOCE)*, 13(2):8, 2013.

[54] P.-H. Tan, C.-Y. Ting, and S.-W. Ling. Learning difficulties in programming courses: undergraduates' perspective and perception. In *Computer Technology*

and Development, 2009. ICCTD'09. International Conference on, volume 1, pages 42–46. IEEE, 2009.

[55] V. Vainio and J. Sajaniemi. Factors in novice programmers' poor tracing skills. In *ACM SIGCSE Bulletin*, volume 39, pages 236–240. ACM, 2007.

[56] A. van Wijngaarden. *Recursive definition of syntax and semantics*. North Holland Publishing Company, 1966.

[57] A. von Mayrhauser and A. M. Vans. *Program Understanding: A Survey*. Colorado State Univ., 1994.

[58] L.-C. Wang and M.-P. Chen. The effects of game strategy and preference-matching on flow experience and programming performance in game-based learning. *Innovations in Education and Teaching International*, 47(1):39–52, 2010.

[59] S. Weinschenk. *100 things every designer needs to know about people*. Pearson Education, 2011.

[60] N. Whitton. Motivation and computer game based learning. *Proceedings of the Australian Society for Computers in Learning in Tertiary Education, Singapore*, 2007.

[61] D. F. Wood. Problem based learning. *Bmj*, 326(7384):328–330, 2003.

[62] P. Wouters, E. D. Van der Spek, and H. Van Oostendorp. Current practices in serious game research: A review from a learning outcomes perspective. *Games-based learning advancements for multi-sensory human computer interfaces: techniques and effective practices*, pages 232–250, 2009.