A Web-Based Environment to Teach Introductory Programming based on a Bi-Directional Layered Notional Machine

Li Sui
Massey University Palmerston North, New Zealand
leesui0207@gmail.com

Jens Dietrich
Massey University Palmerston North, New Zealand
j.b.dietrich@massey.ac.nz

Eva Heinrich
Massey University Palmerston North, New Zealand
e.heinrich@massey.ac.nz

Manfred Meyer
Westphalian University of Applied Sciences, Bocholt,Germany
manfred.meyer@w-hs.de

ABSTRACT

We present a novel browser-based environment to teach introductory programming. This platform combines gamification with peer-to-peer interaction. Students write programs (bots) that play simple board games on their behalf, and can exercise these bots by playing against the bots developed by their peers. The tool is web-based in order to facilitate low-cost delivery and collaboration.

The user interface provides unique tracing features based on a bi-directional layered notional machine. This creates some interesting technical challenges due to the limitations of traditional tracing technologies such as debugger interfaces and continuations. We discuss how we solved these problems, and present several experiments used to validate the scalability of our approach.

The proof-of-concept implementation is based on the SoGaCo platform and the PrimeGame, using Java as the programming language.

1. INTRODUCTION

Improving the performance of students in introductory programming courses remains a challenging problem. While novel educational platforms alone will not be able to solve this problem, there are success stories such as GreenFoot [8] and Scratch [13] that have clearly demonstrated how educational outcomes can be improved by using the right tools.

To be effective, any educational platform must provide attractive features that engage and encourage students, and relate to how they interact with technology and their peers outside the classroom. Animation and gamification are widely seen to be such features, and many tools use them.

In our previous work, we have suggested that the integration of social networking is such a feature as well [4]. In particular, social networking can be used to allow students to collaborate and compete in tournament-like settings when combined with gamification. The engineering of such a platform has to overcome several challenges with respect to security and scalability. On the other hand, such a distributed system approach facilitates the delivery of the platform as a web-based service, mimicking the design and mode of delivery of very successful systems like Google Docs. For the educator, this has the added advantage of low total cost of ownership, in particular freeing him or her from on-site system installations. Furthermore, an approach that is based on server-based services has the inherent advantage that it can be easily adapted to different clients, including browsers on traditional personal computers and custom apps for the various mobile platforms. Our previous work on SoGaCo [4][1] has demonstrated how such a platform can be implemented, and how security and scalability challenges can be addressed.

The basic principle of SoGaCo is that students create bots that play board games on their behalf. Bots are edited in browser-based editors, and submitted to a central server for storage and execution. Bots can be easily shared via unique URLs, and students can use this feature to play against other bots, including bots shared by peers. The SoGaCo framework has several variation points that facilitate different games, programming languages and authentication methods.

The focus on board games facilitates a user interface design centred around a simple notional machine [5, 6]. This machine is based on the visualisation and the terminology of the board game used, displaying game states after each turn computed by a bot. The expectation is that this will support the student in understanding the semantics of the program developed. However, using only a visualisation of the board game is not sufficient as students also need to understand the program states that lead to game states.

The contribution of this paper is a novel type of notional machine developed to address this issue. The rest of this paper is organised as follows: we review related work in section 2, followed by a discussion of the notional machine in section 3. We then summarise the results of some experiments

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

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

used to confirm the technical feasibility of our approach in section 4. This is followed by a brief conclusion section.

2. BACKGROUND AND RELATED WORK

In this section, we discuss some related work on notional machines, reverse debugging and the PrimeGame which we use for our proof-of-concept implementation.

The concept of a notional machine was introduced by du Boulay as "the general properties of the machine that one is learning to control" as one learns programming [5]. He suggested that such a notional machine should be simple, useful and observable. An overview of current work on notional machines is given by Sorva [17]. According to Sorva [17], a notional machine: (1) is an idealized abstraction of computer hardware and other aspects of the runtime environment of programs; (2) serves the purpose of understanding what happens during program execution; (3) is associated with one or more programming paradigms or languages, and possibly with a particular programming environment; (4) enables the semantics of program code written in those paradigms or languages (or subsets thereof) to be described; (5) gives a particular perspective to the execution of programs; and correctly reflects what programs do when executed.

Several studies, e.g. by Chen et al. [3], have shown that many of the problems students have in introductory programming courses stem from misconceptions of the notional machine, especially of those aspects not directly apparent from the program code but hidden within the execution-time world of the notional machine.

Smith and Webb [16] relate students' difficulties in developing and debugging their programs to an inadequate mental model of how the computer works when executing their programs. This also limits the use of program tracing, a key programming skill that novice programmers often struggle with as shown in a multi-national study byLister et al. [10]. A study by Fitzgerald et al. [7] showed that students often struggle in selecting the right "moving elements" to keep track of in tracing which results in information overload and failure to understand program execution. Notional machines have also been looked at from a constructivism perspective, e.g. by Ben-Ari and Yeshno [1], claiming that novice programmers need to form an understanding of the computer matching some normative model, possibly a notional machine operating close to the level of abstraction that the learners focus on.

The idea of navigating through the program execution trace not only in chronological order as the program is being executed but also back and forth as necessary to identify and investigate critical parts of the program has been discussed as *reverse* or *historical* debugging. Lessa and Jayaraman [9] describe JIVE, a visual debugger for Java built on the Eclipse IDE that depicts both the runtime state and call history of a program in a visual manner and offers forward and "reverse stepping" with the ability to jump directly back to any previous point in the execution history in order to observe the object diagram at that point, to better understand program execution and narrow down the cause of program errors. Reverse debugging has also found its way into the GNU Project Debugger[2] and the OCaml debugger[3] developed at INRIA for the programming language Caml offering `step` and `backstep` commands. Microsoft's IntelliTrace also offers reverse debugging functionality as part of the VisualStudio IDE under the term *historical debugging*.[4]

The PrimeGame is a mathematical programming game which has been used in undergraduate computer science teaching since 2003 [12]. Due to its very few and simple rules it can be easily understood and played. However, the complex search space asks for developing some strategy rather than being able to evaluate all possible moves to the end. Beside being used as a vehicle to motivate and activate programming novices, it has also been used in second-year courses to discuss agent models and communication protocols when student groups were allowed submitting a set of (cooperating) bots instead of individual bots competing with each other [11].

3. THE NOTIONAL MACHINE

In this section, we describe the unique features of the notional machine we have developed: its *layered design*, and the *reversibility*.

3.1 The Game Play Layer

The system presented here is based on the SoGaCo platform [4]. SoGaCo allows students to develop *bots* that play simple board games on behalf of the user against other bots. This is based on the assumption that students find it easy to understand the game rules, and develop strategies to play the game encouraged by the social and gaming features of the system. While bots are developed on the client, the compilation and execution happens on the server. This facilitates collaboration and a low total-cost-of-ownership delivery as a service.

The particular game we chose for the proof-of-concept implementation is the PrimeGame [12]. The PrimeGame has a particular simple set of rules that can be explained in minutes[5], and a range of strategies which most students quickly discover and try to implement – ranging from simple cautious and greedy (play the smallest or the largest number on board) to more sophisticated strategies (play largest prime number on board, optimise net gain, look ahead). When students discover and implement new strategies, they often must employ additional language features (conditionals, loops, nested loops and helper functions). This progression therefore encourages learning and supports comprehension of the respective programming language features and the respective algorithmic constructs.

When a bot is developed, the user can test it against other bots. This includes built-in bots of various complexity, or bots shared by other users. The turn-based character of the game facilitates game replay. I.e., the user interface features a game board with media-player like controls that allows the user to replay the game. This functionality is widely used

[2]http://www.gnu.org/software/gdb/

[3]http://caml.inria.fr/pub/docs/manual-ocaml-4.00/manual030.html

[4]http://msdn.microsoft.com/en-us/library/mt228143.aspx

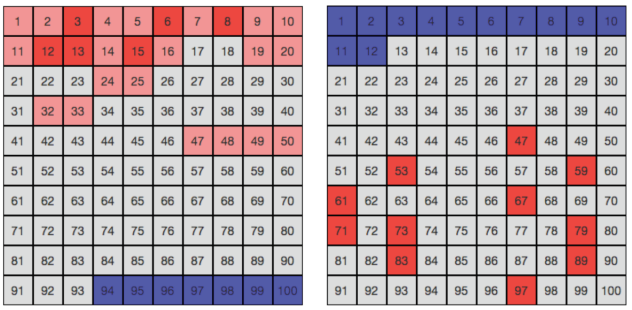
[5]The PrimeGame is a turn-based game played by two players using a simple board that consists of all natural numbers from 1 to 100 (or any other maximum number chosen). A player plays a number that is on the board, and gains this number as points. For each prime factor of the number played still on the board, the opponent gains those numbers as points. Then the number played and its prime factors are removed from the board, and it is the opponent's turn. The player with the highest number of points wins the game.

in electronic board games, and represents a simple, easy to comprehend notional machine. The model used is the game board itself, and therefore requires little abstraction, is easily accessible to all students, and can be used to understand the impact of certain decisions made in the program.

The board visualisation consists of boxes containing numbers, arranged in a 10 × 10 grid. Players are associated with colours (a "red bot" plays against a "blue bot"), and these colours are used to display the game state. Examples of game boards are shown in figure 1. Both games are unfinished as they contain grey fields indicating that these numbers have not yet been played. In the game shown in figure 1 (a) a blue bot uses a simple greedy strategy (select the largest number on board) against a red bot using a cautious strategy (select the smallest number on board). The points gained in a move by the blue player are highlighted with a *dark* blue background, while the points gained by the opponent during this very move due to the prime factor rule are highlighted *light* red. The respective student owning the blue bot can easily see that his strategy is not smart as it will help the opponent to gain many points even when it is not its turn. On the other hand, in the game shown in figure 1 (b), the red bot plays the largest prime strategy. This means that the blue player does not gain any points from the red player's moves[6]. In summary, the game board-based notional machine enables students to quickly comprehend the shortcomings and advantages of certain strategies.

A noteworthy feature of this machine is the ability to undo moves. I.e., the notional machine is *reversible*. This is intuitive and expected by students used to media-player controls and undo functions in editors. It is also easy to implement as the game state after each turn is easy to capture, compact and can be transferred quickly to the client for visualisation.

The shortcoming of this machine is that it only exposes the game state between turns, and treats the computation of the actual turns as black box. This becomes problematic once students try to implement more sophisticated strategies, and start creating complexity they then have problems to understand. Debugging is required to reveal the code hidden in the black box.



(a) Greedy (blue) against cautious (red)

(b) Prime (red) against cautious (blue)

Figure 1: Board state for games played by bots implementing different strategies

3.2 The Bot Debugging Layer

[6]Unless the red bot starts the game, in this case the blue bot will gain 1.

We address this problem by adding a second layer to the notional machine. To understand the execution of the program that computes the next move, students can use debugger-like functionality. This allows students to monitor the execution of their program by inspecting the values of the different variables currently on the stack. The interface we have developed for this can be seen in figure 2. A notable features that distinguishes our approach from a classical debugger UI is that there are no breakpoints where the execution of the program is stopped. Instead, there is a forward control that is used to replay the execution starting at the beginning of a turn. We considered using an alternative visual notional machine like the one suggested by Berry and Kölling [2], or a machine based on dynamic UML diagrams [15]. However, these visualisations are more suitable for object-oriented programming. We do not aim at a particular programming paradigm, and bots developed to play the PrimeGame are procedural by nature as the problems do not lend themselves to object-oriented solutions with stateful entities.

While creating a simple visual debugger is straight forward, there are complications from integrating this into the overall web-based architecture, and with the top-level notional machine.

First, there is the problem of directionality. The game play layer allows students to replay and reverse game play. This is an important feature as students can use it to understand how the game state evolved. The reverse button gives them the change to repeat steps they have problems comprehending. The availability of the reverse function has an impact on the debugger API: the debugger must be able to replay execution starting at each turn.

To achieve this poses challenges, in particular when the logic is implemented on a shared server with limited resources. While it is possible to devise services that return the state of execution (i.e., the state of the Java stack and the associated parts of the heap) for a certain turn in a game played between two bots, this requires complex versioning or locking schemes as the bots may change during a debugging session due to the inherently concurrent character of the platform. Even the use of simple server-based debugging sessions through the Java debugging interface (JDI) or continuation-style APIs [14] is problematic as this requires to keep games in shared server memory in long-running sessions and therefore has a negative impact on scalability.

To solve this problem, we developed a debugger that captures the entire stack and the referenced parts of the heap during the execution of a game (on the server), serialises and compresses this information and returns it to the client. While this comes at a cost - the resources needed to extract, compress and transmit this information, it minimises the number of active computations on the server and leads in general to better server throughput. Some of the experiments discussed in section 4 quantify the overhead.

To extract debug information, we used the byte code instrumentation already built-into SoGaCo and discussed in [4]. The original purpose of this instrumentation was to inject code to monitor user-submitted code to avoid denial of service attacks by enforcing timeouts and memory quota. This is achieved by inserting code via byte code instrumentation that generates notifications around invocations and assignments. An overview of the build and instrumentation

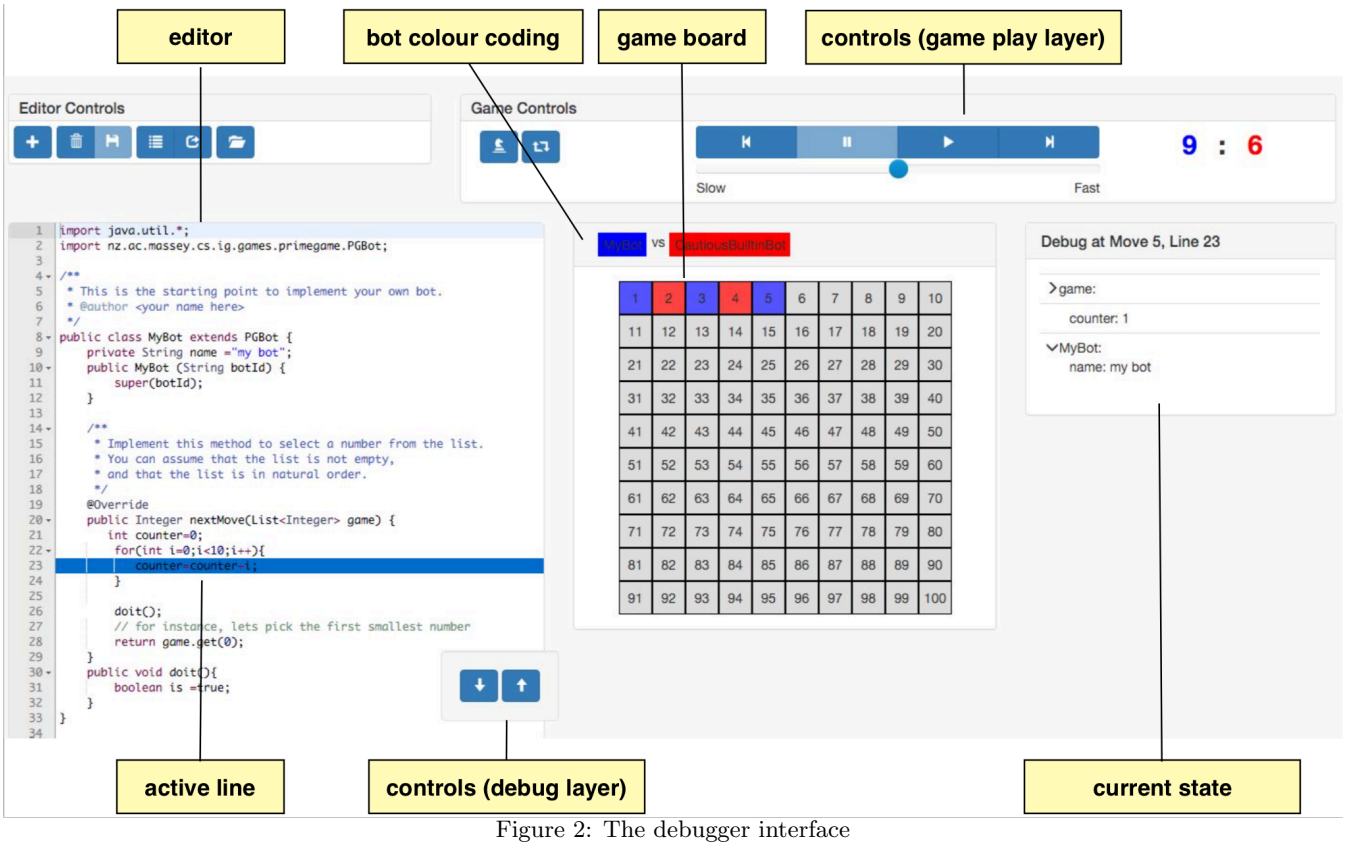


Figure 2: The debugger interface

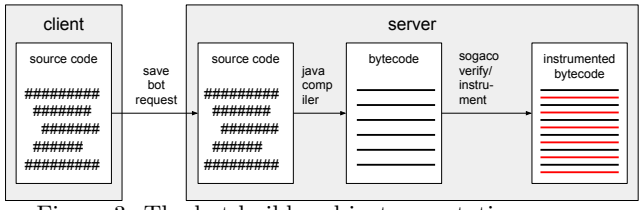


Figure 3: The bot build and instrumentation process

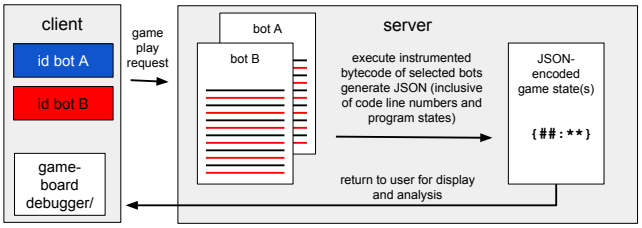


Figure 4: Capturing state during game execution

process is shown in figure 3. Figure 4 depicts how the instrumented code is used to capture game state, to encode this state and return it to the client. This is described in more detail below.

An additional benefit of transferring the complete execution state to the client is that this allows us to build a completely *reversible debugger*. I.e., the user cannot only resume debugging from the beginning of a turn, but (s)he can also step backward within the execution for a particular turn. There is again an educational benefit here: students can use undo functionality they are used to in order to repeat steps of the computation they have problems to understand.

4. VALIDATION

In this section we investigate whether a complete capture of the execution in order to facilitate reversible game replay on the client is feasible.

4.1 Benchmarks

There are several constraints to consider: (1) the time needed to capture and encode state after each state changing instruction; (2) the server memory required to represent the state in order to compose the response; (3) the size of the response that must be transmitted to the client. (2) can be mitigated by streaming results using technologies like server-sent events[7]. However, this comes at the price of keeping HTTP connections open longer, and we therefore did not consider this. We note that (3) can be at least partially addressed by using the transparent (GZIP) compression built into HTTP, and supported by all mainstream browsers and server.

The most important factor that will impact (1)-(3) is the complexity of the bots. Bots that use multiple control structures, and in particular nested loops, will perform more instructions for which states need to be recorded. Also note that a "deep capture" of the objects on the heap is necessary after each state-changing instruction, as we must assume that objects are mutable, and could be changed by executing any instruction.

[7]https://html.spec.whatwg.org/multipage/comms.html#server-sent-events

We use the following three benchmarking bots to take measurements. The *greedy bot* is the bot that always picks the largest number on the board. It is a very simple bot that does not require the use of conditionals or loops, and is a typical starting point for new students. The *smarter bot* is a bot the maximises the net gain (points gained by self minus points gained by opponent through the prime factor rule). The smarter bot beats bots playing the largest prime number, and requires code that uses multiple control structures and helper functions. This is usually the most sophisticated bot students will produce in an introductory programming course. The *black mamba* is a bot produced by a teacher and represents the worst case scenario in terms of complexity. The black mamba uses a sophisticated strategy that looks ahead and tries to simulate the moves of the opponent. This requires the exploration of a very large search space. It is unlikely that students in introductory courses produce a bot like this. More complex bots are likely to be rejected by the server due to enforced memory and CPU quota.

4.2 Encoding Schemes

A critical feature of our system is how state is represented and encoded. Figure 5 depicts the structure of captured state. The execution of a game consists of a series of invokations of the `move(..)` method to compute the next turn. The method itself performs multiple byte code instructions. After each instruction that alters state (in particular, `store` and `invoke` instructions), a memory snapshot is created via instrumentation. This snapshot includes the stack and parts of the heap. This includes temporary variables as well as fields. For reference variables encountered, references are traversed up to a certain depths to capture the relevant part of the heap. This *capture depth* can be configured, the default value is two. This has the consequence that the debugger allows users to inspect objects in simple data structures like lists and maps, but not in nested lists or lists within maps. We also simplify the object structure by using a logical "flat" representation for common data structures. For instances, maps are represented as simple lists of key-value pairs, and lists ae represented by their elements, ignoring intermediate objects representing buckets (in hash maps) or entries (in linked lists) as these details are not useful for the target audience. Many mainstream IDEs use a similar approach in their debuggers. While the capture depth value can be increased, it means that the space required to capture state increases for games that use "deep" data structures. We use JSON to serialize the captured state as this facilitates the interaction with web clients.

We experimented with different encodings and compression methods.

- **baseline encoder (BE)**. A simple encoder that maps the entire state up to a certain depth to a JSON (nested) object.
- **custom encoder (CE)**. A better encoder improved by a custom representation of lists of numbers as intervals. For example, a consecutive number array [0,1,2,3,4,5] is represented as an interval ["0-5"].
- **dictionary index method (DI)**. State is recorded in maps that for each variable associate tuples consisting of a turn number, a line number and an iteration count (for loops) with JSON-encoded variable values. Note that such records are only created if we encounter instructions that can modify the value or state of the respective variable.

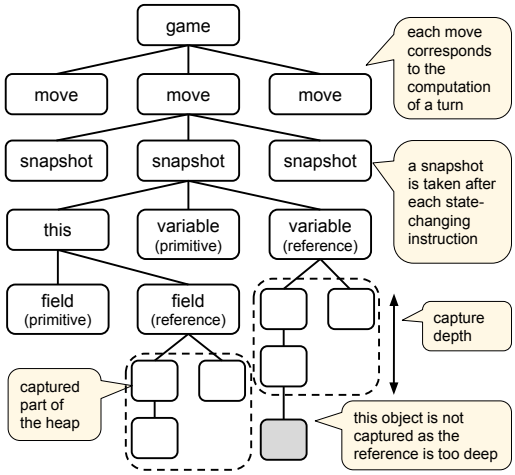


Figure 5: Structure of captured memory data

- **edit distance method (ED)**. This is based on the idea of using delta compression where a snapshot is represented by the changes with respect to the previous snapshot. This compression is only used if variables are added or removed, which corresponds to code features like variable declarations, or block exits.
- **tree edit distance method (TED)**. This is similar to the edit distance method, but also encodes changes (add, remove and update) in branches of the tree.

4.3 Results

We executed several experiments to validate the encoding strategies described in section 4.2. These experiments were executed on a MacBook Pro 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 used the benchmarks described in section 4.1. We also included a "null scenario" where we executed the game without capturing state in order to create a baseline for the runtime.

For each experiment, the respective bot class is instantiated twice, and the bots play against each other. State is captured only from one bot to simulate a game executed on the SoGaCo server. We took the following measurements: (1) the overall execution time of the bot monitored (2) the size of heap space needed to represent the captured and encoded state[8] (3) the size of the captured state when exported to the file system (4) the size of the file compressed using the popular zip utility. The last measurement is relevant as this is a good estimate for the network load for server to client data transfer if the server is configured to use compression.

The respective results are summarised in table 1. The first row represents the "null scenario". The results indicate that the size of the data produced is very sensitive to both the complexity of the bot monitored and the method used

[8]For this purpose, we used the SizeOf object size estimation library(http://mvnrepository.com/artifact/com.carrotsearch/java-sizeof)

Table 1: Results

encod. meth.	bot	time (ms)	memory (kb)	file size (kb)	compr. size (kb)
null	greedy	1	n/a	n/a	n/a
	smarter	7	n/a	n/a	n/a
	mamba	34	n/a	n/a	n/a
BE	greedy	1	13	16	4
	smarter	297	11,264	11,980.8	119
	mamba	6,361	148,480	155,852.8	1,945.6
CE	greedy	1	5	8	4
	smarter	277	5,120	6,963.2	82
	mamba	5,154	113,664	120,115.2	1,740.8
DI with CE	greedy	1	3	4	4
	smarter	259	63	66	4
	mamba	5,970	1,024	2,048	172
ED with CE	greedy	1	5	8	4
	smarter	260	1,024	1,536	82
	mamba	6,385	70,656	74,649.6	2,251.8
TED	greedy	5	11	12	4
	smarter	626	1,024	1,228.8	86
	mamba	29,946	30,720	32,768	2,150.4

to capture and encode state. However, with sensible choices, it is possible to encode the entire state and therefore to support reversible debugging on the client while facilitating a server architecture that supports high scalability. The most suitable method is clearly the directory index method.

We also experimented with increasing the capture depth. For space reasons, we cannot provide full details here. Bots like greedy are not affected by this. For the black mamba using the optimal DI encoding method, the compressed file size changes from 172 kb (capture depth = 2, see table 1), to 389 kb when setting the capture depth to 3 and stabilises after this.

5. CONCLUSION

In the previous sections we have focused on the technical architecture of our platform that, on a conceptional level, consists of two interrelated notional machines. Students learning to program build mental models of notional machines and execute those models, in their mind, to reason about the system. Sorva [17] discusses the challenges faced in positioning a mental model and the related notional machine on the right level of abstraction. Our proposition is that our two connected notional machines, one at game level, one at program level, assist with this challenge. Students can seamlessly switch between levels and freely step forwards and backwards in either, allowing them to shift focus and repeat single steps as often as required to form understanding. In future work we need to track how students interact with our platform and how this relates to the development of their mental models.

6. REFERENCES

[1] M. Ben-Ari and T. Yeshno. Conceptual models of software artifacts. *Interact. Comput.*, 18(6):1336–1350, 2006.

[2] M. Berry and M. Kölling. The state of play: A notional machine for learning programming. In *Proceedings ITICSE'14*. ACM, 2014.

[3] C.-L. Chen, S.-Y. Cheng, and J. M.-C. Lin. A study of misconceptions and missing conceptions of novice java programmers. In *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS'12)*, 2012.

[4] J. Dietrich, J. Tandler, L. Sui, and M. Meyer. The primegame revolutions: A cloud-based collaborative environment for teaching introductory programming. In *Proceedings ASWEC'15*. ACM, 2015.

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

[6] B. Du Boulay, T. O'Shea, and J. Monk. The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies*, 14(3):237–249, 1981.

[7] S. Fitzgerald, G. Lewandowski, R. McCauley, L. Murphy, B. Simon, L. Thomas, and C. Zander. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Comput. Sci. Educ.*, 18(2):93–116, 2008.

[8] P. Henriksen and M. Kölling. Greenfoot: combining object visualisation with interaction. In *Proceedings OOPSLA'04*. ACM, 2004.

[9] D. Lessa and B. Jayaraman. Explaining the dynamic structure and behavior of java programs using a visual debugger. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE)*. ACM, 2012.

[10] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppäla, B. Simon, and L. Thomas. A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bull.*, 36(4):119–150, 2004.

[11] M. Meyer. The PrimeGame reloaded: Finding the right balance between cooperation and competition in undergraduate computer science classes. In *Proceedings EDULEARN11*. IATED, 2011.

[12] M. Meyer and J. Fendler. The PrimeGame: Combining skills in undergraduate computer science programmes. In *Proceedings INTED'10*. IATED, 2010.

[13] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: programming for all. *Commun. of the ACM*, 52(11):60–67, 2009.

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

[15] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Pearson, 2004.

[16] P. A. Smith and G. I. Webb. Reinforcing a generic computer model for novice programmers. In *Proceedings of the 7th Australian Society for Computer in Learning in Tertiary Education Conference (ASCILITE'95)*, 1995.

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