Learning Concurrency Concepts while Playing Games

Cornelia P. Inggs, Taun Gadd and Justin Giffard
Computer Science, Dept. of Mathematical Sciences, Stellenbosch Univ., Private Bag X1, 7602 Matieland, South Africa

Keywords: Concurrency, Game, Learning, Programming.

Abstract: We think that people will find it easier to learn concurrency concepts if they can play a game that challenges the player to solve puzzles using the same techniques required by a programmer to develop concurrent programs. This article presents two such games in which multiple threads of control are represented by multiple avatars that can perform actions concurrently in a game environment. The player controls the avatars by specifying a sequence of actions for each avatar to execute using a block-based visual syntax, independent of programming language. The avatars execute their actions in the game environment, showing the effect of every action.

1 INTRODUCTION

People typically find it more difficult learning how to develop concurrent programs than sequential programs. Many programmers start by learning how to develop programs with one thread of control which execute a sequence of instructions. Such sequential programs may contain branching conditions, which result in multiple possible execution sequences, but during a particular run of a sequential program only one sequence of instructions is followed, as determined by the input given to the program.

In a concurrent program multiple tasks or logical threads of control can be in progress at any instant. The instructions of the different threads of control can be executed in parallel, i.e., simultaneously on separate execution units, or the instructions can be interleaved and executed on a single execution unit as a sequence of instructions, but in this case the particular sequence is not only determined by the input given to the program. In a concurrent program the order in which instructions are executed, whether simultaneous or in sequence, is nondeterministic and is influenced by conditions outside of a user's control, such as the scheduler, the load of the system, and contention for system resources.

This nondeterministic execution order of instructions, of which the effect is not immediately visible, and the large number of possible execution sequences that a concurrent program can have as a result, makes it difficult to write error-free concurrent code or to reproduce an error that occurs only for some execution sequences. Concurrency related errors occur when there are dependencies between instructions, which require sharing of information between different threads of control, which in turn requires instructions to be executed in a certain order—or not simultaneously with other instructions—to preserve correctness.

Programmers that develop concurrent programs therefore need to write code that will be correct for any input data as well as any order in which instructions are executed. This requires an understanding, not only of how to divide work between threads of control, but also of how to share information between threads of control and how to use synchronisation techniques to share information safely—and without introducing further errors such as deadlocks.

One way to learn a new concept is to learn the same concept in the context of a familiar setting and then transfer the knowledge to another setting. For the introduction of sequential programming concepts, a wide range of computer games have been developed and are available online, from games that provides a visual block-based syntax for specifying sequences of actions (Google, 2015; MIT; Lightbot), to games that require specifications in a programming language (MIT, 2015; HopScotch, 2016; CodeCombat, 2016). Game-oriented tutorials for learning concurrency concepts have also been developed, but those that were found available online require programming language knowledge (Hudeček and Pokorný, 2016). We think a game with a visual block-based syntax for specifying sequences of actions and visual feedback of their effect in the game would make it easier for people to learn concurrency concepts. In fact, we

Inggs, C., Gadd, T. and Giffard, J.
Learning Concurrency Concepts while Playing Games.
DOI: 10.5220/0006374705970602
In *Proceedings of the 9th International Conference on Computer Supported Education (CSEDU 2017) - Volume 1*, pages 597-602
ISBN: 978-989-758-239-4

think it should be possible for a child of eight or nine years old, mature enough to think abstractly, to learn concurrency concepts.

2 RELATED WORK

A number of applications for teaching sequential programming concepts have been developed. The applications most related to our work are those that use a game-oriented approach and a visual syntax independent of programming language. Blockly Games and Lightbot are examples of such games and served as inspiration for our game (Google, 2015; Lightbot).

Blockly Games is a set of games that has been developed for children to learn sequential programming concepts such as conditional and control-flow statements (Google, 2015). Users are provided with a game environment that contains objects and Google's Blockly editor, which contains interlocking blocks (Google, 2016). Users are given a problem to solve or a task such as: move the turtle object a number of steps forward to reach the patch-of-grass object. The blocks represent coding concepts and, for a sequence of blocks, the Blockly library can return syntactically correct code in the language chosen from the supported list of languages.

Lightbot provides, similar to Blockly Games, a game environment with an avatar and an editor with action blocks (which represent conditional or control-flow statements) that can be placed in sequence to manipulate the avatar (Lightbot). It provides a sequence of predesigned puzzles to solve, which increases in difficulty and is divided into groups where each group starts with the introduction of a new block (code concept). Each puzzle only requires the use of blocks that have been introduced in a previous group or at the start of the current group. The simplicity of Lightbot's syntax places even more emphases on solving problems using coding concepts (without having to learn the syntax of a programming language) than the interlocking block-based games.

There are also non-game-oriented programming tools available that provide a block-based syntax similar to Blockly Games, such as Scratch (MIT), App Inventor for Android devices (MIT, 2015), and Hopscotch (HopScotch, 2016). These tools do not give a user tasks to solve, but provide the user with an empty execution environment, where they can create their own programs by placing objects in the execution environment and sequences of blocks in the editor to control when and how to add or move objects or when to play sound clips, for example.

On the other hand, games that require users to write code in the syntax of a real programming languages also exist. Code Combat, for example, gives users problems to solve or a task to complete in a game setting, but requires the user to write the sequence of actions for the avatar to execute, in the syntax of Python, JavaScript, HTML, CSS, jQuery, or Bootstrap (CodeCombat, 2016).

There are three game-oriented approaches, designed to teach concurrency concepts, that are related to our work. The first, Deadlock Empire, is the only game-oriented application for teaching concurrency concepts that we could find available to play online (Hudeček and Pokorný, 2016). The Deadlock Empire requires a user to complete a sequence of tutorials/challenges; in each the player is presented with two threads and the objective of the player is to schedule the instructions of the threads such that the code breaks; revealing, for example, a race condition or a deadlock. This game is particular good at showing the effects of nondeterminism and the errors that can result when an incorrect execution order is chosen, but it is programming language specific: one has to be able to read code similar to the C syntax.

The second is a series of activities that have been designed to help students build an intuition about synchronisation by managing "railway semaphores" using an open source train simulation game (OpenTDD, 2014). It was designed to be a lab project; the students download the OpenTDD simulator as well as a set of three activities, which include OpenTDD scenario files and game files and once they have installed the required software, it typically takes them less than an hour to complete the activities (Marmorstein, 2015). The last is a proposal for an educational game that could be used to teach concurrency concepts. A user would be shown a rectangular grid with boxes, obstacles, markers, and robots and would be required to program the robots to move all the boxes to their corresponding markers. An initial version with alphabet characters has apparently been implemented, but the design is not complete and, as discussed in the Conclusion, the authors still need to investigate whether it would be possible to teach all the concurrency concepts using their game (Akimoto and de Cheng, 2003).

In the next section we'll discuss the concepts that a programmer needs to understand to develop concurrent programs and in the following section we discuss two games that have been developed to teach these concepts. The games require the players to solve puzzles using a simple set of blocks, independent of a programming language, to specify the sequence of actions the avatars in the game should execute. When the avatars execute their actions, the player has visual

feedback on the effect of the actions; for example, one can see when two avatars are in deadlock, waiting for each other to execute an action before continuing.

3 CONCURRENCY CONCEPTS

As explained in the introduction, concurrency errors due to nondeterministic orderings of instructions occur when threads of control share information. The concepts that should be understood can therefore be classified under three categories: dividing work among multiple threads of control, sharing information between the threads of control, and synchronising the sharing of information.

Concepts related to the division of work include, for example, whether tasks are assigned to threads of control before execution starts (static load balancing) or at runtime (dynamic load balancing) and the performance impact of a particular division of work; a good division of work into independent tasks can result in speedup while a poor division of work can result in load imbalance and therefore less than ideal speedup.

When the work can not be divided into independent tasks, the threads of control need to share information, and programmers therefore need to understand the different ways in which information can be shared, i.e., via either synchronous or asynchronous message passing or via access to shared memory.

The final category includes all the concepts related to ensuring correct execution of multiple threads of control that work together to solve a problem and need to share information either regularly or occasionally. Programmers need to understand the effect of nondeterministic orderings, synchronisation techniques, error behaviour due to the absence of synchronisation (race conditions), error behaviour due to incorrect synchronisation (deadlock, livelock, starvation), and finally the performance impact of synchronisation, e.g., waiting for a shared resource (such as a lock).

4 THE GAME

Two games were implemented: Parallel Bots (see Figures 1 and 2) and Parallel Blobs (see Figure 3). Both games consist of a game environment that contains multiple avatars and an editor that provides a block-based visual syntax independent of programming language.

The **game environment** is set up with a challenge, such as: program the two avatars to turn on all the lights in the game environment as fast as possible. The game environment of Parallel Bots is an isometric grid similar to LightBot; it renders the grid to a JavaScript canvas, which can contain one or more objects such as light switches, boxes, machines, shelves, walls, and avatars. The game environment of Parallel Blobs was created using Unity and consists of one or more two dimensional rooms with walls between them and contains one or more objects, such as mailboxes, keys, doors, walls, bridges, completion pads, and avatars. The game environment is displayed while a player specifies a sequence of actions in the editor for each avatar. When the play button is clicked the avatars execute their actions until they are either blocked or have completed their sequence of actions.

Figure 1: Parallel Bots. In this puzzle the avatars have to execute the lightbulb action (toggle a switch) on each circle in the game environment to turn its light on. The screenshot was taken while the avatars were busy executing their actions.

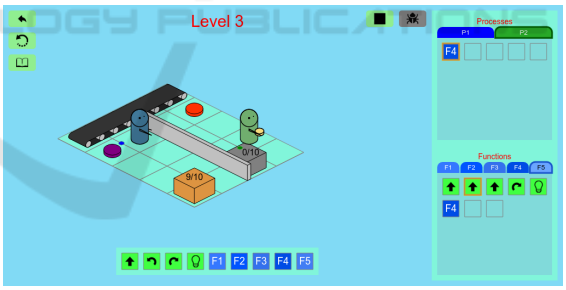

Figure 2: Parallel Bots. In this puzzle the conveyor belt is only active when both avatars are in position (on their disks), the one wants to send an object on the belt, and the other is waiting for an object.

The **editor** contains multiple windows: one for each avatar—where a player can program an avatar by specifying the sequence of actions for that avatar—and extra function windows where a player can specify common sequences of actions than can then be called from any of the other editor windows. Only two editor windows are displayed at any time, the window for the currently selected/executing avatar (blue/P1 in Figure 1) and the window for the currently selected/executing function (F1 in Figure 1). During execution, the action being executed is highlighted in

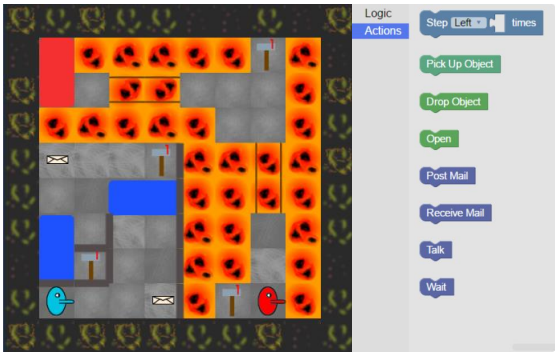

Figure 3: Parallel Blobs. This puzzle requires the avatars to send messages to each other to draw bridges over the lava. The editor is currently in block-choosing mode and all the action blocks are shown.

yellow.

The sequence of actions for an avatar is specified by using the blocks provided by the game. In Parallel Bots, a limited set of action blocks are provided and each action is represented by only one block; they are: walk forward, rotate 90°, pick up/put down/toggle the switch of the object on the facing block, or lock/unlock the object on the facing block. Parallel Blobs provides a Blockly editor that contains all the core blocks plus new blocks that were created specifically for Parallel Blobs: Step, Pick up/Drop an object, Open a door, Send/Receive mail, Talk, and Wait (see Figure 3). The block-based syntax of Parallel Blobs is not as simplistic as that of Parallel Bots: there are more blocks, many blocks can take parameters, and some actions are represented by more than one block.

Both games have a series of challenges that increases in difficulty. The challenges are divided into groups where each group starts with the introduction of a new action block (code concept). Each challenge can be solved using only known blocks, i.e., blocks that have been introduced in any of the previous groups or at the start of the current group.

The next section compares the concepts the players learn by solving the puzzles to the concepts that a programmer needs to develop concurrent programs, and compares the games with relevant approaches.

4.1 Comparison: Concepts Covered

4.1.1 Dividing Work

In both games multiple threads of control are represented by multiple avatars that can perform actions concurrently. Like threads of control, the multiple avatars can execute independently or share information and they can execute the same sequence of actions or different sequences of actions.

For some of the puzzles currently provided each avatar has specific tasks to execute and for others the player can decide how to divide the tasks among the avatars, but all the current puzzles requires the player to assign specific tasks to avatars before execution starts and program each avatar to complete only its tasks; this is called **static partitioning**. **Dynamic partitioning** allows the assignment of tasks during execution based on current workload. A puzzle that requires dynamic partitioning can be added by changing one of the current puzzles as follows: a box contains objects of different sizes that need to be taken to one of two other locations based on their size. Taking the bigger objects to their location takes longer than taking the smaller objects to their location and the next object to be removed from the box is determined by a random number generator. When the avatars are programmed to take the next object as soon as they have taken the previous object to its location, the partitioning of tasks would be clearly dynamic, because the order in which the objects are removed is determined at runtime by the random number generator.

When work is divided among threads of control and not all the partitions take the same amount of time to execute, a load imbalance can occur. **Load imbalance** is usually more of a problem when work is statically partitioned than when it is dynamically partitioned during runtime, because during runtime the current workload of a thread of control can be taken into account.

The effect of load imbalance is clearly visible in the game; in fact it is more visible than when you run a concurrent program. When a concurrent program executes you typically don't know when threads have completed their work and are idle, waiting for other threads to complete their work; you often only know when the last thread completes and the program terminates. In the game, it is immediately obvious when an avatar has completed its sequence of actions and are waiting for another avatar to complete his.

On the other hand, a player can see the difference in completion time (in number of time units) for a particular puzzle if the puzzle is first solved with one avatar and then with more than one avatar that can divide the work among them. If two avatars can divide the work to solve a specific puzzle evenly between them, they could take half the time that one avatar takes to complete the puzzle; which would represent ideal **speedup**.

Waiting for an avatar to complete a task does not only happen when there is a load imbalance. It can also happen when avatars need to share objects or information and one avatar has to wait for another avatar to provide the object or information required.

4.1.2 Sharing Information

Message passing can be performed asynchronously or synchronously. Asynchronous message passing is depicted by a mailbox in Parallel Blobs and a shelve in Parallel Bots. It is asynchronous, because players can try to add objects to (or remove objects from) the mailbox/shelve and then continue with other actions, whether the avatar with whom it is sharing the object is busy executing something else or busy waiting for an object at (or adding an object to) the mailbox/shelve.

In Parallel Blobs synchronous communication is represented by thin walls: two avatars can communicate if they stand on either side of a thin wall, but if they stand further away or on either side of a thick wall they can not hear each other. In Parallel Bots synchronous communication is depicted by a conveyor belt; an object can only be sent to another avatar via a conveyor belt if the one avatar is standing on a disk on the one side of the conveyor belt and trying to put an object onto the conveyor belt and the other avatar is standing on a disk on the other side of the conveyor belt and trying to receive an object.

Shared Memory can also be used to share information. In Parallel Bots, a box represents a shared memory location in which light bulbs can be placed. Avatars have to take turns to add objects to/remove objects from a shared box; they should not access the box simultaneously. In Parallel Blobs, shared memory is represented by narrow tunnels and narrow bridges, which should be used by only one avatar at a time.

In both games **mutual exclusive access** to shared objects are obtained via a shared key, which emulates a lock (e.g., semaphore). In Parallel Bots a **race condition**, which occur when access to a shared object is not synchronised and simultaneous access is obtained, is emulated by a light bulb shattering into pieces. In Parallel Blobs, the absence of synchronising access to a narrow bridge/tunnel, results in two avatars colliding.

Two or more avatars are in **deadlock** if all of them are waiting for an action to occur that can only be caused by one of the other avatars in the set. In Parallel Blobs, two avatars will, for example, be in deadlock if they need the same two keys to complete a task, the one avatar has picked up the one key, the other avatar has picked up the other key, and each is waiting for the second key to become available. In Parallel Bots, two avatars will be in deadlock if they're on either side of the conveyor belt, which represent synchronous communication, and both want to receive an object before they can continue.

Two or more avatars are in **livelock** if they repeat the same sequence of actions over and over again, but no progress is made. In the games this could occur, for example, when two avatars indefinitely hand an object or send a message back and forth.

Finally, we need to discuss nondeterminism. **Nondeterminism** is present in the games, but it is mostly determined by the order in which the player decides to place actions and the order in which the Round Robin scheduler executes the actions of the different avatars; currently the avatars are executed using a Round Robin scheduler where one action is executed per time quantum. During the execution of concurrent programs on a real system other programs also compete for memory bandwidth, cache usage, CPU time, etc. We would like to increase the effect of nondeterminism in the games by adding other scheduling options as well as a random number generator that can influence the scheduler's decisions, so that nondeterminism due to parameters outside of the programmer's control can be emulated. The debug/step option can also be adjusted to allow a separate step button for each avatar to allow finer grained control over the possible interleavings during debugging.

4.2 Comparison: Relevant Approaches

The focus of The Deadlock Empire is to show the effect of different scheduling orders; it shows the effect of the execution order a player chooses for two specific C code segments and the player is challenged to find an execution order that will result in an error. The focus of Parallel Bots/Blobs is problem solving using concurrent programming techniques without any knowledge of a programming language. Different sequences in Parallel Bots/Blobs may result in different sequences, but with the current implementation a specific set of sequences will result in the same interleaving if left unchanged between runs. The same single step scheduling control as provided to a player of The Deadlock Empire can be provided to a player of Parallel Bots/Blobs by changing the debug/step option so that a separate step button is provided for each avatar.

The lab project described in (Marmorstein, 2015) was designed to help students build an intuition about synchronisation while Parallel Bots/Blobs were designed to help the player build and intuition about all the core concurrency concepts and, instead of following the approach of a single lab project it follows the approach of a game that provides challenges to solve.

The educational game proposed in (Akimoto and de Cheng, 2003) seem to follow similar ideas as Parallel Bots/Blobs, but since their robots only move boxes to specified locations on a grid, it is not clear how they

will support some of the concurrency concepts, such as asynchronous message passing, for example.

5 CONCLUSION

We think that people will find it easier to learn concurrency concepts if they can play a game that challenges the player to solve puzzles using the same techniques required by a programmer to develop concurrent programs; techniques for dividing work, sharing information, and synchronising the sharing of information among multiple threads of control.

Two games are presented in this article and in both, multiple threads of control are represented by multiple avatars that can perform actions concurrently. The player controls the avatars by specifying a sequence of actions for each avatar to execute using a block-based visual syntax (i.e., the player programs the actions of the avatars instead of using game controls) and the avatars perform their actions in a game environment that has been set up with a puzzle.

A detailed comparison between the concepts a player of the game learns when solving the puzzles and the concepts a programmer needs to develop concurrent programs is included in the article; which makes it clear that even advanced concurrency concepts can be used to solve puzzles in the game environment. The main difference between a concurrent program's threads executing instructions and the avatars of the game executing their sequences of actions is the visibility of their effect. When a concurrent program has a race condition or is in deadlock, it is often difficult to detect what went wrong and when it happened, while the effect of every action is clear in the game environment. Such immediate visual feedback typically makes learning easier.

Another benefit of the game environment and its block-based syntax independent of programming language, is that by providing puzzles at levels of increasing difficulty, young children as well as older students can enjoy solving them. The speed at which players progress through the levels and the highest level that can be completed will most likely differ for players of different ages and backgrounds. We are in the process of conducting a study to evaluate which concepts people of different age groups can understand and whether understanding the concepts in the game settings does indeed make it easier to learn how to develop concurrent programs.

Finally, two limitations that have been identified are being addressed in the next version: the scheduler will be updated so that it is adjustable from the execution environment and include a random option that can emulate nondeterminism due to parameters outside of the programmer's control; and a puzzle builder will be added so that new puzzles can be added by dragging and dropping objects onto an empty game environment, instead of updating source code directly.

REFERENCES

Akimoto, N. and de Cheng, J. (2003). An educational game for teaching and learning concurrency. In *Proceedings of the 1st International Conference on Knowledge Economy and Development of Science and Technology*, pages 34–39.

CodeCombat (2016). A classroom in-a-box for teaching Computer Science. https://codecombat.com/. Accessed 2017-01-30.

Google (2016). Blockly: A library for building visual programming editors. https://developers.google.com/blockly/. Last updated 2016-10-14.

Google (2015). Blockly Games: Games for tomorrow's programmers. https://blockly-games.appspot.com/. Last updated 2015-01-17.

HopScotch (2016). Coding made for you. https://www.gethopscotch.com/. Last blog entry 2016-11-04.

Hudeček, P. and Pokorný, M. (2016). The Deadlock Empire: Slay dragons, master concurrency. https://deadlockempire.github.io/. Accessed 2017-01-30.

Lightbot. Solve puzzles using programming logic. https://lightbot.com/. Accessed 2017-01-30.

Marmorstein, R. (2015). Teaching semaphores using... semaphores. *Journal of Computing Sciences in Colleges*, 30(3):117–125.

MIT. Scratch: Create stories, games, and animations. https://scratch.mit.edu/. Accessed 2017-01-30.

MIT (2015). MIT App Inventor: A beginner's introduction to programming and app creation. http://appinventor.mit.edu/explore/. Accessed 2017-01-30.

OpenTDD (2005-2014). An open source simulation game based upon Transport Tycoon Deluxe. https://www.openttd.org/en. Accessed 2017-01-30.