Chapter 3

Concurrency Education with Sonic Pi: "Hear" and "Play" Mistakes and Misconceptions in Multithreaded Programs[1]

Concurrency is a complex to learn topic that is becoming more and more relevant, such that many undergraduate CS curricula are introducing it in introductory programming courses. This chapter investigates the combined use of Sonic Pi and Team-Based Learning to mitigate the difficulties in early exposure to concurrency. Sonic Pi, a domain-specific music language, provides great support for "playing" with concurrency, and "hearing" common problems such as data races and lack of synchronization among different concurrent threads. More specifically, the chapter focuses on students' misconceptions regarding concurrency in Sonic Pi and compares them to those arising in traditional concurrent programming languages. In addition, it preliminary explores knowledge transfer from Sonic Pi to C/C++.

3.1 Introduction

Background and Motivations Concurrent programming is a technique in which two or more executions threads start, run in parallel or in an interleaved fashion through context switching, and complete in an overlapping time period by managing access to shared resources. Multithreading can be compared to a pianist playing a piano with both hands. Instructions run in parallel on a single processing unit. The threads may work on different instructions (such as the

[1]This chapter is based on [50, 51]. An extended version of [51] is under revision in the journal INFEDU.

pianist playing two separate musical pieces), but work with the same resources and processing power. Concurrent programs can execute multiple tasks at once, e.g., writing a file on disk while appending new items. Concurrency, however, introduces nondeterminism: the exact order in which tasks are scheduled is not known in advance. Furthermore, the interleaving of instructions of different execution threads may yield an exponential number of possible schedules. It is often the case that some schedules will lead to correct outcomes and others will not. Thus, it becomes necessary for programmers to express constraints to prevent the system from allowing schedules that yield incorrect outcomes.

In the past, due to its complexity, concurrency was typically introduced at advanced levels of computer science (CS) curricula. However, due to its growing importance, e.g., event-driven programming and multi-core computing, several universities are now rethinking their approach and starting to introduce concurrency in introductory courses, in some cases adopting simplified concurrency models [173, 58]. Despite this, students often struggle to understand concurrency, which can be particularly challenging when introduced early in the curriculum. Consequently, there is a need for research aimed at effectively introducing concurrency and supporting students in building and refining their viable mental models.

In this chapter we investigate the use of Sonic Pi to mitigate the difficulties in early exposure to concurrency. Sonic Pi [1] is a domain-specific programming language and code-based music tool where musical concepts, and specifically those related to multi-threading, are aligned with programming ideas. The Sonic Pi programming tasks have been collocated within a collaborative framework based on the Team-Based Learning (TBL) pedagogy []. They focused on code comprehension and code composition and targeted common misconceptions about concurrent programming.

The approach, based on an interdisciplinary pedagogy for early exposure to concurrency by combining music and programming activities, has been applied to two teaching experiments for undergraduate CS students, more precisely 130 first-year students in the context of an annual first-year course that focuses on teaching students the fundamentals of computer architecture design for modern microprocessors and 54 third-year students in the context of a third-year, second-semester course that focuses specifically on the design and analysis of algorithms for concurrent and distributed systems.

Research Questions Based on previous research on concurrency education in traditional programming languages, our research questions are:

- Do misconceptions and mistakes in Sonic Pi correspond to misconceptions and mistakes in traditional concurrent programming languages? Does the use of Sonic Pi help the students face common misconceptions and mistakes in concurrent programming? Is this more helpful for early-stage programmers?

- Can we apply knowledge transfer from Sonic Pi to more traditional programming languages such as C and C++? What is the effect in terms of learning outcomes?

Contributions The contribution of this chapter is twofold. Firstly, it presents an introductory approach to concurrency based on Sonic Pi, aimed at undergraduate students, and shares all the detailed information and materials so that other lecturers can replicate the activities. This teaching format is based on our extensive research and classroom experimentation over the past three years, which has provided a solid foundation. Secondly, the chapter explores students' misconceptions about concurrency that emerged during the experiments. By collecting and analyzing team responses and tasks, we delve into students' problem-solving strategies, comparing them to experiences with those based on traditional programming languages.

Design The choice of the Sonic Pi programming language is due to several reasons. As mentioned in [2], *"Sonic Pi was designed to teach a large number of computing concepts covered in the new UK computing curriculum introduced in September 2014. Examples of these concepts are conditionals, iteration, variables, functions, algorithms, and data structures. We also extend beyond these to provide educators with an opportunity to introduce concepts that we believe will play an increasingly important role in future programming contexts such as multi-threading and hot-swapping of code."* Sonic Pi offers distinct advantages over traditional computing methods to facilitate the early introduction of multi-threading. The inherent concurrency of music creation (e.g., several instruments that synchronize on a rhythm and melody) and the simplicity of the tool can provide a natural and immersive learning experience for concurrency. In particular, live coding and auditory output, two key features of Sonic Pi, provide great support for "playing" with concurrency, and "hearing" data races and other common mistakes. To our knowledge, this study represents the first attempt to use Sonic Pi for these learning goals at the university level.

The proposed teaching format is based on our previous research and experiments started in the academic year 2019/2020. We started by experimenting with the use of Sonic Pi with TBL to increase learning motivation and curiosity among freshmen by introducing them to advanced topics [182].

Building on this experience, we further refined and adapted the teaching format in the academic year 2021/2022 for third year students enrolled in the Concurrent Programming and Distributed Algorithms (CP[2]) and Informatics for Creativity, Didactics and Dissemination (ICDD) courses. These courses provided an ideal platform to explore concurrency and its application in creative contexts. In these experiments, the primary learning objective was to introduce students to the concept of multi-threading in Python. However, we approached this goal by starting with concurrency in Sonic Pi and then moving to Python using the `python-sonic` library. This interface allowed students to combine the multi-threading capabilities of Python with the functionality of

[2]Programmazione Concorrente e Algoritmi Distribuiti in Italian.

Sonic Pi within a Python environment. For the students enrolled in the ICDD course, the experiment served an additional purpose. It provided an opportunity to examine the pedagogical objectives and didactic concepts of the experiment itself.

The two units proposed in this thesis, and experimented during the 2022/23 academic year, are based on our previous experience. They represent the culmination of our efforts to introduce concurrency in a creative way and to address misconceptions in concurrency education More specifically, we focused on the goal of introducing students to the concept of concurrency through a series of short practical tasks, each designed to address specific misconceptions about concurrency and basic programming concepts, allowing students to gradually build their understanding while addressing specific misconceptions, including variable scope, function calls, parameter passing, race conditions, correctness and synchronisation goals. Some tasks focused on program comprehension, while others involved code writing. Program comprehension tasks [83] are a constructionist approach to teaching programming, where the learner interacts with an artifact representing the program, for example a piece of code. Through this interaction, the learner is stimulated to build and refine their viable mental model of the underlying NM.

The program comprehension and composition tasks in Sonic Pi have been designed taking into account common misconceptions observed both in our previous teaching experience and in the literature on concurrency education, in particular, those arising from the mixing of concepts from concurrency and the underlying computational models [173], which seem to be particularly challenging for students.

Experimental Evaluation To assess the effectiveness of the activity in terms of engagement and appreciation we considered students' overall perceptions through a post-questionnaire; to evaluate the effectiveness in terms of learning outcomes, we considered TBL data, in particular team responses to the TBL tasks.

In addition, we wanted to gather preliminary empirical evidence on the transferability of the concurrent programming knowledge, skills, and abilities that students acquire in our Sonic Pi-based learning environment to other traditional programming languages, such as C. Knowledge transfer is crucial for effective learning, but it can pose challenges and lead to misconceptions for several reasons, such as differences in the underlying notional machines (NM) of the programming languages considered. To investigate knowledge transfer, we designed a written test consisting of three exercises in either C or pseudocode, focusing on concurrency aspects covered in the music code-based domain. The test was administered one week after the experiment.

Plan of the Chapter Section 3.2 discusses related work while Section 3.3 introduces some preliminary notions about Sonic Pi and TBL. Section 3.4, 3.6 and 3.7 provide all the details of the design of the TBL units (RAT Quiz and Team app). In Section 3.8 we introduce the teaching experiments and we present an aggregate analysis of data collected via quizzes, tests,

and questionnaires. Section 3.9 discusses findings and implications. In Section 3.10 we address some conclusions and future work.

3.2 Related Work

This section presents related work, by surveying approaches combining music and coding, approaches for teaching concurrency, focusing on the most common misconceptions related to concurrency, and finally introducing knowledge transfer.

3.2.1 Music and Coding in CS Education

This section discusses the use of live coding and musical approaches in CS education, along with supporting tools and languages for teaching concurrency. Live coding involves the real-time manipulation of running programs to generate live auditory or visual effects, with the coding process becoming an integral part of the performance [20]. From a pedagogical perspective, live coding provides immediate feedback and enhanced interactivity, making it a valuable tool for teaching computer programming. In particular, live music coding allows abstract programming concepts to be presented in a more natural way through music.

In recent years, live music coding has gained prominence in interdisciplinary contexts, particularly in the field of STEAM education [26]. This approach has proven successful in teaching programming in both introductory and advanced programming courses [69]. Furthermore, the integration of music and CS has been found to enhance students' appreciation, motivation, and engagement in the learning process [74]. The authors of [76] highlight the effectiveness of domain-specific programming platforms in promoting intrinsic motivation and positive attitudes towards learning CS. In [69], the authors present an approach to teaching design patterns and other programming topics (data structures, grammars, parsing, and formal proofs) using a music composition project. A summer camp is presented in [108], designed with a "Code Beats" approach, where students learn fundamental programming concepts to make music using a domain-specific tool that provides immediate feedback and hints for learning. The experiment seemed to increase student interest, motivation and engagement. Finally, in [148] Scratch music coding capabilities are used to teach basic programming concepts. There are a number of educational code-based music creation tools, such as JythonMusic[3], EarSketch[4], TunePad[5] and Sonic Pi[6]. Sonic Pi, developed in 2012 with the UK's new primary and secondary computing curriculum in

[3] https://jythonmusic.me/
[4] https://earsketch.gatech.edu/landing/#/
[5] https://tunepad.live/
[6] https://sonic-pi.net

mind, has proved effective in introducing live music coding to primary schools [1]. It also appears to be effective in promoting positive attitudes towards programming among middle school students [134]. Furthermore, the use of multimedia computer contexts in teaching introductory programming across educational levels has shown positive effects on pass and retention rates [163].

3.2.2 Concurrency Education

In the field of concurrency education, several studies have focused on analyzing programming errors made by students. One study examined errors in specific concurrent assignments [106], while another conducted a phenomenographic study to understand how students develop concurrent programs [106]. Additionally, the analysis of final written exams from concurrent and operating systems courses aimed to identify common misconceptions among students [174]. Finally, empirical research at the secondary school level has investigated students' perspectives on the correctness of concurrent programs [88, 18].

Concurrency poses challenges as it involves non-determinism, making it difficult to test program correctness by running them. Unlike sequential programming, concurrency requires a more formal and reasoned approach, demanding students to develop viable mental models of concurrent program execution. In addition, students are expected to possess a solid understanding of fundamental concepts like variable scope, parameter passing, aliasing, references, and pointers, which are considered *threshold concepts*—challenging concepts that require significant development or revision of specific areas of the mental model. Mastery of these concepts enables students to recognize connections with other programming concepts [22, 173].

Given the complexity involved, concurrent programming has traditionally been covered in advanced courses that require strong programming skills, such as operating systems courses or other advanced electives. However, the increasing importance of concurrency, as discussed in the previous section, has prompted the CS education community to address concurrency in introductory courses. This has led to a reevaluation of curricula and the emergence of new pedagogical approaches [173].

Over the past few decades, several tools have been developed to aid students in learning about concurrency. Eludicate [81] and Atropos [107] are examples of tools that allow students to capture and visualize concurrent object-oriented execution. However, these tools lack active interaction with the visualization, which diminishes engagement and learning impact, as argued by Sorva [171]. To address this limitation, other tools have been introduced, such as The Deadlock Empire[7], an online educational game specifically designed for learning concurrency. Progvis [175] is another tool that enables visualization of the interaction between concurrency and fundamental programming concepts like scope, parameter passing, and references.

[7] https://github.com/deadlockempire/deadlockempire.github.io

In addition to these tools, older examples include Linda [30] and Multi-Logo [142]. Linda is a coordination language that supports concurrency, and Multi-Logo is a concurrent extension of Logo, which was experimented in primary schools where students had to control simple robotic devices constructed from LEGO® bricks.

In principle, formal concurrency models may be considered as possible NMs to be used to reason on concurrent executions abstracting away from the syntax of a particular programming language. Process calculi such as CSP [79] and CCS [121] and graphical concurrency models such as Petri nets [133], are well-known abstract models used to represent concurrent computations as mathematical objects and to reason about their properties [96].

For instance, in [17], Mordechai Ben-Ari adopts a logic-based approach based on automata, temporal properties and model checking to reason about properties such as mutual exclusion, starvation, and deadlock interleaved with practical examples of concurrent programming libraries in Java and Ada.

3.2.3 Common Misconceptions in Concurrent Programming

Misconceptions in concurrent programming can often lead to subtle and hard-to-debug issues. In this section, we consider misconceptions related to basic and advanced concepts in concurrent programming.

Many students struggle with synchronization problems, and part of this may lie in their understanding of the prerequisites that are crucial to concurrent programming. In fact, students may struggle not only with concurrency concepts but also with threshold concepts about the underlying computational model.

As discussed, e.g., in [173, 174], students often struggle with the scope of variables, mistakenly believing that local variables are shared between threads or concurrent function calls. This misconception stems from a failure to distinguish between different instances of function calls and to recognize that each instance has its own set of local variables.

Other misconceptions include the belief in global locks as protectors of code rather than protectors of specific resources, which stems from a non-viable mental model that focuses on code rather than data. Additional common misconceptions concern correctness [89], believing that a concurrent program only needs to work most of the time rather than all of the time. They may rely on trial and error for debugging, ignoring the concept of interleaving, which is difficult for many students to grasp. It has also been found that many students experience the program in terms of the program text rather than its dynamic execution [174].

In addition, students find it difficult to identify concurrency problems and formulate synchronization goals [19]. However, once synchronization goals are identified, they generally succeed in implementing solutions that meet those goals. However, in one study [174], students struggled to

associate shared data with appropriate synchronization primitives, such as using multiple locks to protect the same instance in different situations.

Students often believe that using synchronization primitives, such as locks or semaphores, ensures thread safety [75]. Students need to understand the importance of correct synchronization placement, avoiding race conditions, and understanding the semantics of different synchronization mechanisms.

Another misconception relates to memory models, where students often assume sequential consistency [75]. This misconception may stem from concurrency courses that focus on context switching on a single CPU and neglect other scenarios involving weaker memory models.

A further common misconception about concurrency, unrelated to the basic concepts, is that adding concurrency to a program will automatically improve performance [177]. In fact, concurrency introduces additional complexity, so it is important for students to understand that performance improvement depends on several factors, such as the nature of the problem, the hardware, and the efficiency of the concurrency design.

Lastly, although it is not necessary to know object-oriented programming (OOP) to work with concurrency, it was observed that a non-viable mental model of OOP can lead to misconceptions in identifying shared data and synchronization mechanisms since some OOP concepts are also relevant to concurrency (e.g. confusing class and instance, which is similar to confusing instances of a function call, values, and references, etc.).

Finally, it is important to note that these misconceptions may vary according to programming language, individual experience, and educational context. Nevertheless, investigating these misconceptions can help to better know student understanding and improve future concurrency education.

3.2.4 Knowledge Transfer

Knowledge transfer refers to *the application of skills (or knowledge, strategies, approaches, or habits) learned in one context to a novel context* [9]. In particular, we refer to the knowledge domain transfer [15], which is the process of applying previously acquired knowledge and skills from one context to another (in the same or a different discipline). In our case, concurrent programming concepts and concurrent problem-solving strategies facilitate the learning and adoption of concurrency in another programming language. However, transferring knowledge from a language such as Sonic Pi to more traditional textual languages such as C and C++ can be challenging due to differences in their NMs. We are not currently aware of any existing research that explores this specific scenario, and further research is needed to understand the effectiveness and challenges of knowledge transfer in this learning context.

Extensive research into the transition from block-based to text-based languages has highlighted

differences in syntax, mental models, misconceptions, program comprehension, and learning outcomes [93]. However, current research seems to point in the direction of a positive transfer between block-based and text-based languages [188].

Learning a second programming language can be challenging [68], even for experienced developers, because of the need to adapt mental models to new language features [161]. However, in some cases, programmers moving from one language to another, such as from C# to Ruby, chose to start from scratch, ignoring prior knowledge, which mitigated the effects of cross-language interference [161].

3.3 Preliminaries

Our proposal relies on a combination of Sonic Pi and Team-Based Learning (TBL) as the teaching format. The resulting learning experience combines hands-on programming activities with collaborative learning. In this section, we focus on the preliminary notions at the basis of the proposed approach, namely, Sonic Pi and Team-Based Learning.

3.3.1 Sonic Pi and Concurrency

Sonic Pi is a domain-specific language for manipulating synthesizers through time. It can also be viewed as a code-based musical creation and performance tool, where each musical concept corresponds to a programming idea. The music domain allows for a pedagogy that focuses on the problem (concurrency) rather than the programming language, proposing real-world examples that are inherently multithreaded. From this perspective, Sonic Pi seems to be naturally constructionist[8] (the *"building material"* [128] to learn concurrency), as Papert's computational thinking stresses the importance of the computer as a powerful meta-tool for *"making the abstract concrete"* [105].

From a technical point of view, Sonic Pi is based on the Ruby programming language, inheriting its simple syntax. It offers an intuitive and user-friendly integrated development environment (IDE) for creating music. Furthermore, Sonic Pi provides auditory output feedback, allowing users to hear the sound generated by their programs in real time. This audio feedback, combined with textual program output and compiler error/warning messages, offers immediate and tangible results, enhancing the learning experience. These characteristics make Sonic Pi a language with a steep learning curve, extremely effective in adapting to different learning goals and environments, and in tailoring topics to students' interests and attitudes [1].

[8]Constructionism is a learning theory created by Seymour Papert and based on the educational paradigm of Piaget's constructivism. Its most famous application is Logo.

Sonic Pi is officially distributed for Microsoft Windows, Mac OS X and Raspberry Pi OS (but there are also unofficial distributions for Linux). Compared to other live coding languages, it is very easy to install, as it provides an all-in-one installer and does not require a separate interpreter. Once inside Sonic Pi, there is no need to route audio to channels: just the command "play" followed by a note number is enough to create the sound. Sonic Pi does not use the usual pitch classifications found in live coding, such as the frequency swing in SuperCollider, but assigns a number to each key of a standard piano [70]. Sonic Pi takes advantage of the speed of

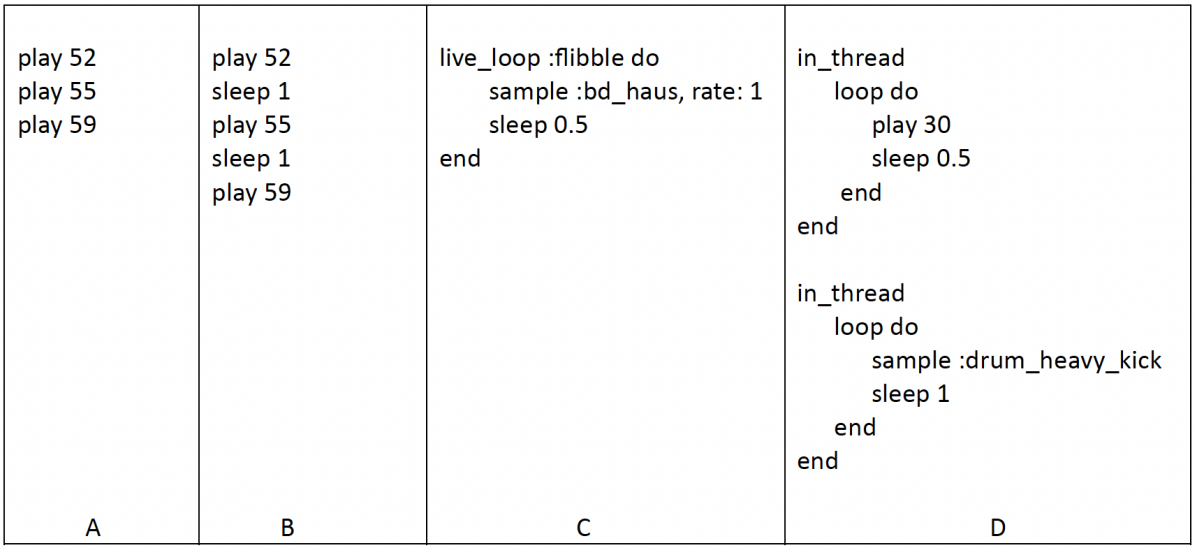


Figure 3.1: Some examples of Sonic Pi programs and instructions.

modern processors in assuming that a sequence of instructions, as those depicted in Fig. 3.1 (A), are likely to be executed so quickly in succession that they will be perceived as a chord, and not as an arpeggio. An arpeggio form can be achieved by "sleeping" the current thread for a number of seconds as in Fig. 3.1 (B). The notion of sleep is similar to that of the standard POSIX sleep operation that suspends execution for the specified time. Sonic Pi 2.0 has introduced special semantics that avoid drifting due to delays (thread scheduling and invocation of the POSIX sleep operation) [3].

Finally, Sonic Pi supports various APIs, allowing users to interact with other platforms and expand its capabilities, such as the `python-sonic` interface.

Due to these factors, Sonic Pi is an ideal tool for teaching both the fundamentals of computer programming, such as iterations, selections, functions, and data structures, as well as advanced concepts like concurrency and scope rules. One of the unique advantages of Sonic Pi is its ability to synchronize and audibly perceive different musical instruments, allowing learners to "feel" local and global objects in action.

The language offers several interesting concurrent features, such as the `in_thread` and `live_loops` control constructs, which are interpreted across multiple concurrent threads. The `live_loop` construct is the key to mastering live coding with Sonic Pi. For instance, consider the following program in Fig. 3.1 (C). Here we create a bass drum beating by repeating the sample `:bd_haus` forever. Thanks to hot-swapping code live loops can be redefined on-the-fly while still running. The `in_thread` construct resembles traditional thread-spawning operations in languages such as C. For instance, consider the following program in Fig. 3.1 (D). Here the MIDI note 30 is played at the same time as the sample `:drum heavy kick` with half a second between each onset.

These programming abstractions provide an intuitive introduction to concurrency, even for novice programmers who are guided by sound and perception. The resulting programs can therefore be validated with a sense of rhythm and melody.

Sonic Pi provides the `cue` and `synch` functions to create synchronized music patterns between threads and/or live loops and avoid drifting effects. The cue sync mechanism is very similar to the `notifyAll()` and `wait()` methods of the Condition objects of other concurrent programming languages, such as Python and Java. Finally, Sonic Pi uses a lock-based synchronization mechanism, providing the `get` and `set` functions to prevent race conditions. It also utilizes a global memory store called *Time State*, where threads and live loops can share data.

Overall, Sonic Pi's combination of auditory feedback, intuitive concurrency features, and synchronization mechanisms makes it an effective tool for introducing concurrency and other programming concepts.

3.3.2 Team-Based Learning (TBL)

To support a collaborative learning environment, we used the TBL teaching methodology. This approach encourages collaboration between team members, and healthy competition between teams, and enriches the learning environment with gamification elements. All these characteristics made TBL consistent with Papert's constructionist idea of the importance of students' social and effective involvement in the construction of a computational artifact ([105]). Collaborative learning has been widely explored in CS education and there is extensive literature on its benefits. In particular, TBL has been shown to be effective in preventing student dropout and improving exam pass rates in CS1 courses [163]. In addition, a qualitative study has shown that TBL is highly rewarding and engaging for students enrolled in CS1 courses [86].

TBL is a strategy that enables students to follow a structured process to improve their engagement and the quality of their learning. It consists of modules that can be taught in a three-step cycle: pre-class preparation, in-class Readiness Assurance Process (RAP), and application-focused exercise (Team APP). More specifically, TBL has five essential components plus an optional peer evaluation phase. It begins with individual study outside the classroom, followed by a multiple-

choice test (RAT), first individually and then in teams who must agree on answers. The RAP phase ends with immediate feedback (usually through scratch cards, which add a playful component to the learning), a possible team appeal and a class discussion with the instructors. This is followed by the team application (APP), which is an open-book task where each team works on the same exercise and has to give an answer at the same time. Finally, the teams discuss and compare their solutions in plenary. This phase can be done with different discussion techniques, such as the *gallery walk*, where each team presents its solution in a kind of poster session.

Technically, we designed RAT quizzes on the course Moodle page. Immediate feedback was given at the end of the team quiz. In addition, we proposed a digital gallery walk using Padlet[9], an online bulletin board tool that allows team solutions to be shared with the whole class.

3.4 Learning Content & Material

The design focused on the goal of introducing students to the concept of concurrency through a series of short practical tasks, each designed to address specific misconceptions about concurrency and basic programming concepts, allowing students to gradually build their understanding while addressing specific misconceptions, including variable scope, function calls, parameter passing, race conditions, correctness, and synchronization goals. Some tasks focused on program comprehension, while others involved code writing. Program comprehension tasks [83] are a constructionist approach to teaching programming, where the learner interacts with an artifact representing the program, for example, a piece of code. Through this interaction, the learner is stimulated to build and refine their viable mental model of the underlying NM.

In this section, we outline the learning content we developed for the TBL units. Specifically, we first present the preparation material in Section 3.5, then the RAT quizzes in Section 3.6 and the team application tasks in Section 3.7.

3.5 Pre-Class Preparation

To ensure adequate preparation and familiarisation with the TBL methodology and the Sonic Pi language, we introduced students to these topics one week prior to the in-class activity. We assigned the following pre-class materials:

- Sonic Pi Tutorial[10] (a fully integrated tutorial that provides a comprehensive introduction to Sonic Pi, assuming no prior knowledge of coding or music).

[9] https://it.padlet.com
[10] https://sonicpi.net/tutorial

- Multimedia footage of live coding performances, available on the Sonic Pi website.
- Slides explaining the TBL methodology.

All preparatory materials were uploaded to the course Moodle page for easy student access. We estimated that students would need about 2-3 hours of individual study to complete the preparation before the class sessions.

3.6 RAT Quiz

We designed an identical RAT quiz for both activities, consisting of five multiple-choice questions that assessed participants' understanding of Sonic Pi language syntax and concurrency concepts. Following the principles of TBL, the quiz was first completed individually and then retaken in teams. The results of these individual and team RATs were collected for information purposes only. As they served as a preparatory tool for the subsequent team application phase and were completely independent of the research question, they were not statistically validated or included in the data analysis. Nevertheless, the results suggest a general trend towards improved performance in teams compared to individual attempts.

The five questions of the Rat quiz are discussed in the rest of the section.

3.6.1 Quiz 1: Hear thread creation

Consider the following program:

```
live_loop :foo do
   play 60
   sleep 1
end
```

When the program runs, you hear a basic beep every beat. Change the note 60 to 65 in the editor without stopping the program. What happens?

- It raises a RunTime error.
- It forks the main thread and creates two new ones. One will play 60 and the other 65. This is live coding.
- No effect on execution. This is live coding.
- It changed automatically without missing a beat. This is live coding.

In Quiz 1, we highlight a distinguishing feature of Sonic Pi, i.e., the dynamic management of code updates, a key point for live coding sessions. Among the multiple choices, only the last answer is correct. Indeed hot code-swapping ensures that the behavior of the thread is automatically updated. This feature is at the basis of live coding in Sonic Pi.

3.6.2 Quiz 2: Hear interleavings

Consider the following code:	What happens if you run it?

`live_loop :foo do`
`  play 50`
`  sleep 1`
`end`
 	
`sample :drum_cymbal_open`
	• All the code after the loop is not executed.
	• The loop and the sample are executed simultaneously.
	• The drum cymbal open sample is played while the loop is "asleep".
	• The loop will repeat 50 times and then play the sample.

In Quiz 2, the attention is focused on the difference between sequential and concurrent execution flow. Here the second answer is the correct one. Indeed, the `live loop` construct starts a concurrent thread which repeatedly plays note 50 while the main program continues to the `sample`.

3.6.3 Quiz 3: Hear the differences between threads and loops

Consider the following code:

```
live_loop :foo do
   sample :ambi_choir
   sleep 0.5
end

in_thread do
   sample :ambi_drone
end
```

What happens if you run it?

- The sample in the `live loop` command is played repeatedly, while the sample in the `in_thread` command is played only once.
- It is not possible to execute both a thread and a live loop concurrently (runtime error).
- Both the live loop and the thread sample are played infinitely.
- Only the live loop will be executed.

In Quiz 3, the attention is focused on the difference between threads and iterative task. The `live loop` command starts a concurrent thread which repeatedly executes its body. The `in_thread` command requires an explicit loop inside its body to repeat a command more than once. Therefore, the first answer is the correct one.

3.6.4 Quiz 4: Listen to data races

Consider the following code:

```
live_loop :setter do
   set :foo, rrand(70, 130)
   sleep 1
end

live_loop :getter do
   puts get[:foo]
   sleep 0.5
end
```

What are the Set and Get functions of the Sonic Pi meant for?

- They allow threads to access a shared resource in a thread-safe way, but mutual exclusion is not guaranteed.
- They allow threads to access a shared resource in a thread-safe way, with guaranteed mutual exclusion.
- They are used to produce non-deterministic program behaviour.
- They are used to manipulate objects whose scope is restricted to a single thread or function.

In Quiz 4, we stress the importance of using thread-safe read/write operations in program with multiple threads. The commands `set` and `get` have been introduced to atomically modify data structures in Sonic pi. Therefore, the second answer is the correct one.

3.6.5 Quiz 5: Avoid Drifting

Given the following code:

```
live_loop :foo do
   use_synth :prophet
   play 20
   sleep 8
   cue :f
end

sleep 0.3

live_loop :bar do
   sync :f
   sample :bd_haus
   sleep 0.5
end
```

Are the two live loops synchronized?

- The two live loops are out of phase due to the 0.3 sleep between the two live loops.
- The two live loops are synchronised because the i-th iteration of the live loop `:bar` is synchronised with the i-th iteration of the loop `:foo` via sync, which waits for the `cue :f` event.
- The code is incorrect and will not be executed.
- The two live loops are not synchronised because the `get` and `set` methods are not used.

In Quiz 5, we stress the importance of time synchronization when reproducing audio signals via multiple threads. Due to possible delays in the scheduling of different threads, multiple threads executing different music samples repeatedly may get out of synch after few iterations, i.e. the resulting program can be affected by the drifting problem. To avoid this problem, Sonic Pi provides synchronization operations between thread groups. More specifically, the commands `cue` and `sync` enforce a rendez-vous synchronization in between the `:foo` and `:bar` threads. Therefore, the second answer is the correct one.

3.7 Team Application

In this Section we provide an overview of the team application tasks we developed for the TBL units. Tasks 1, 2 and 3 were designed for the CA abd CP units, while tasks 4, 5, 6 and 7 were specifically designed for CP students. The team app tasks included a variety of question formats, including multiple-choice questions and open-ended tasks that required students to implement Sonic Pi code. Each exercise targeted specific misconceptions related to concurrent programming. We focused on a range of concurrent concepts, such as data races and synchronization mechanisms, as well as fundamental programming concepts like variable scope and function

calls. The evaluation criteria are based on the rubric presented in Table **??**. It is important to

Table 3.1: Learning Objectives for Concurrency Management

Objective	**Level 1**	**Level 2**	**Level 3**	**Level 4**
Recognizing a shared resource	Does not recognize the presence of shared resources in the exercise	Recognizes the presence of shared resources but does not handle concurrent access correctly	Recognizes the presence of shared resources and implements synchronization mechanisms to ensure concurrent access	Recognizes the presence of shared resources, effectively implements synchronization mechanisms, and optimizes performance
Recognizing synchronization goals	Does not recognize the need for synchronization between processes or threads	Recognizes the need for synchronization but implements inadequate or complex solutions	Recognizes the need for synchronization and implements suitable and understandable solutions	Recognizes the need for synchronization, implements innovative and optimized solutions
Recognizing the correct visibility scope of variables	Does not understand the concept of visibility scope of variables in concurrent context	Partially understands the concept of visibility scope but makes errors in managing shared variables	Understands the concept of visibility scope and properly manages shared variables among threads	Fully understands the concept of visibility scope and implements advanced solutions to manage shared variables
Recognizing race condition	Does not encounter race conditions and fails to resolve them	Identifies race conditions but only partially resolves the problem	Effectively manages race conditions in most cases	Successfully avoids race conditions and implements preventive solutions
Recognizing possible deadlock situations	Does not recognize deadlocks and does not resolve them	Identifies deadlock situations, but with some difficulty	Avoids deadlock situations and manages them effectively most of the time	Successfully avoids deadlock situations and implements effective preventive solutions

remark that, in presence of multiple execution threads, the traditional concepts of variable scope become more complex. In addition to global and block/function local scope, it is also necessary to consider thread local variables and the interplay among all of them.

By embedding concurrent problems within these fundamental programming concepts, we aimed at highlighting the challenges and complications that arise when students do not have a clear understanding and a viable mental model of concurrency and sequential execution.

3.7.1 Task 1: Data Races and Variable Scope, 10 minutes

A

```
in_thread do
  use_synth :piano
  x = 40
  10.times do
    x += 4
    sleep 0.5
    play x
  end
end

in_thread do
 use_synth :kalimba
  x = 40
  10.times do
    x -= 4
    sleep 0.5
    play x
  end
end
```

B

```
x = 40
in_thread do
  use_synth :piano
  10.times do
    x += 4
    sleep 0.5
    play x
  end
end

in_thread do
  use_synth :kalimba
  10.times do
    x -= 4
    sleep 0.5
    play x
  end
end
```

C

```
x = 40
in_thread do
  use_synth :piano
  10.times do
    x += 4
    sleep 0.5
    play x
  end
end

x = 60
in_thread do
  use_synth :kalimba
  10.times do
    x -= 4
    sleep 0.5
    play x
  end
end
```

Voting cards: Which answer is true?

- In B and C, different threads operate on a common resource and the result depends on the order in which the different threads execute their instructions;
- In A and C, different threads operate on a common resource and the result depends on the order in which the different threads execute their instructions;
- In all three programs, there are no resources shared between threads;
- In A and C, there are no resources shared between threads. The result depends on the order in which the instructions of the different threads are executed.

Task 1 is based on of three Sonic Pi codes, namely A, B, and C, and on some ”voting cards” questions. Each code comprises two threads that access a global variable `x` without any synchronization mechanism. Students were required to select the correct answer from the following options. In script A, variable `x` is thread local, whereas in script B and C `x` is global. Therefore,

only A does not present data races as specified in the first answer. Furthermore, in B both threads simultaneously update `x` starting from note 40. The data race can be heard as perturbations of note 40. In C the second assignment eventually overwrites the first one and thus, after a sequence of perturbations of note 40, the program first jumps to the higher note 60 and then the sequence of perturbations continues from that note. The task specifically targets misconceptions related to data races and the management of global shared resources in a concurrent programming model.

3.7.2 Task 2: Data Races and Function Calls, 10 minutes.

```
x = 40

in_thread do
  use_synth :piano
  10.times do
    x += 4
    play x
    sleep 0.5
  end
end

in_thread do
  use_synth :kalimba
  10.times do
    x -= 4
    play x
    sleep 0.5
  end
end
```

```
x = 40

define :foo do |x|
  x -= 4
  play x
end

in_thread do
  use_synth :piano
  10.times do
    x += 4
    play x
    sleep 0.5
  end
end

in_thread do
  use_synth :kalimba
  10.times do
    foo x
   sleep 0.5
  end
end
```

```
x = 40

define :foo do
  x -= 4
  play x
end

in_thread do
  use_synth :piano
  10.times do
    x += 4
    play x
    sleep 0.5
  end
end

in_thread do
  use_synth :kalimba
  10.times do
    foo
    sleep 0.5
  end
end
```

Voting cards: Which answer is true?

1. Mutual exclusion mechanisms are in place in all programs.
2. In a program there is only one possible race condition in read/write but not related to multiple writes.
3. In programs where the “foo” function or procedure is present, these cannot be done check race conditions.
4. None of the previous answers.

Task 2 consists of three Sonic Pi programs and some “voting cards” questions. Students were required to identify potential data races. The task aimed at addressing both race conditions and fundamental programming concepts such as variable scope, function calls, and parameter passing. To identify a data race, students needed to comprehend the scope of variables and function calls within a concurrent scenario. In all three programs, there exists a shared resource referred

to as x. The first program features two threads that access x without any form of mutual exclusion or synchronization mechanism. The second program involves the definition of a function with a single formal parameter. This function is called 10 times by the second thread, with the variable x passed as an argument by value. Consequently, each function frame possesses its own local variable named x. However, as the function is called, the variable x is accessed without any synchronization mechanism in place, which can potentially lead to data races. In the third program, a function without parameters is defined and subsequently called 10 times by the thread. Each time the function is invoked, it accesses the global variable x without any synchronization mechanism in place.

3.7.3 Task 3: Sleep and Data race, 10 minutes

```
use_synth :piano

note=[52,55,59,40]
i=0

define :foo do |x|
  in_thread do
     play note[x]
  end
end

3.times do
  foo i
  i+=1
end
```

Voting cards: Which answer is true?

(1) The program has no race conditions.
(2) The program creates only one thread.
(3) The program plays the notes in the "`note`" list in sequence.
(4) The program plays the E minor chord.

Gallery walk:
Describe in detail the behavior of the script with particular pay attention to the changes to the value of the variable "`i`" and to the possible sequences of notes play.

Task 3 presents a multiple-choice question that requires students to reason about pausing execution and data races. The table shows the Sonic Pi code for this task, the "voting cards questions", and the "gallery walk". The code consists of two global variables: an array of notes and a counter variable initialized to zero. There is also a function with a parameter `x`, which defines a thread to play the x-th note from the array. A loop calls the function three times and increments the counter variable by one. The array is global and shared between the threads. However each thread uses the same array, i.e., potential race condition on a shared data structure, but with a different index, i.e., different cells of the array. Therefore, without any `sleep` invocation to introduce delays in thread execution, the audible output of the program is the E minor chord, i.e., the latter answer is correct.

3.7.4 Task 4: Thread-safeness, 10 minutes

```
A

a = (ring 60, 57, 65)

in_thread do
  10.times do
    a = a.sort
    print a
    sleep 1
    play a
  end
end

B

in_thread do
  10.times do
    a = a.shuffle
    sleep 1
  end
end
```

Gallery walk:
Implement a thread-safe solution for codes A and B using inter-thread synchronization (hint: use Sonic Pi thread-safe variables and methods).

Task 4 is an open-ended task requiring students to identify the synchronization goal and implement a thread-safe solution. The task involves a ring variable, which can be seen as a type of linked list, shared by two threads in a non-thread-safe manner. One thread sorts the ring, then sleeps for 1 second, and finally plays the notes from the ring. The other thread shuffles the ring and then sleeps. Each thread repeats this process 10 times in a loop.

The current implementation suffers from a race condition, resulting in non-deterministic behavior. Sometimes the thread plays the sorted ring, while other times it does not. The misconception lies in the identification of the synchronization goals, specifically the shared resource accessed by both threads. Students were tasked with identifying the synchronization goals and implementing a thread-safe solution to address the race condition. Three possible solutions of task 4 are shown in in Figure 3.2. re details on the solutions will be given in Section 3.8.3.1.

3.7.5 Task 5: Locking and Synchronization, 10 minutes

```
set :add, 40

live_loop :producer do
  note = set :add, get[:add]+20
  play note
  sync :wait
  sleep 0.5
end

live_loop :consumer do
  sync :add
  note = set :add, get[:add]-20
  play note
  cue :wait
  sleep 0.5
end
```

Voting cards: Which answer is true?

(1) There exists an execution in which a thread fails and never obtains a resource (starvation).
(2) There exists an execution in which a thread eventually obtains a resource.
(3) There exists an execution in which the threads fail, and block each other, e.g., waiting for a resource or message (deadlock).
(4) The program does not presents the above mentioned behaviours.

Task 5 consists in a multiple-choice question, requiring students to analyze the lock mechanism and identify the presence of a deadlock. The task involves a shared resource, represented by the variable `:add`, which is accessed by multiple threads in a safe manner using the set-get operators of the Sonic Pi time state. The task is designed to address misconceptions regarding synchronization mechanisms based on message passing, specifically the set-get and cue-sync operations in Sonic Pi.

Despite the thread-safe use of the shared variable, a deadlock situation arises as both threads synchronize using `cue` and `sync` on messages `:add` and `:wait`, but in a wrong order, i.e., both threads remain blockes on the corresponding `sync` invocation.

3.7.6 Task 6: Thread-local scope, 20 minutes

```
x = ???__1__???
use_synth :beep

define :f do |x|
  define :g1 do |y|
    x = y + ???__2__???
    play x
  end
  play x
  sleep 2
  g1 x
end

define :g do |n|
  n = n + ???__3__???
  play n
end

define :h do
  x = x + ???__4__???
  play x
end

define :j do |x|
  x = x + ???__5__???
end

define :p do |x|
  use_synth :chipbass
end

define :q do ???__V__???
  x = x + ???__6__???
end

define :z do
  x = x + ???__7__???
  play x
end

in_thread do
  2.times do
    play x
    sleep 2
    x = x + ???__8__???
  end

  f x
  sleep 2
  g x
  sleep 2
  h
  sleep 2
  j x

  cue :tick
  sync :tick3
  ???__A__???
end

in_thread do
  sync :tick
  2.times do
    ???__B__???
    x = x + ???__9__???
    play x
    sleep 2
  end
  cue :tick2
end

in_thread do
  sync :tick2
  p x
  q ???__W__???
  z
  print x
  sleep 2
  cue :tick3
end
```

Determine the sequence of notes 50 54 58 62 66 70 74 78 82 86 by replacing ???__i__???? with the appropriate values/instructions.
Also determine the instructions (or none) to write instead of ???__A__?? and ???__B__?? to play notes 74 and 78 with the dark_ambience synth and note 86 with the beep synth.

Task 6 involves completing the Sonic Pi code using three threads and parameter passing functions to generate an ordered sequence of audible notes. Understanding thread synchronization and

scope rules for variables and synthesizers is crucial to solving the task. In particular, synthesizers are global but thread-local, whereas variables can be shared between threads and be local within a function. This task requires a higher level of understanding from students. Success in these tasks requires mastery of skills and the ability to apply theoretical knowledge creatively.

3.7.7 Task 7: Master/slaves Synchronization, 20 minutes

Given the following melody (theme from Super Mario Bros.):

```
play_pattern_timed([nil,nil,:e5,:ds5,:d5,:b4,nil,:c5,
                    nil,:e4,:f4,:g4,nil,:c4,:e4,:f4,
                    nil,nil,:e5,:ds5,:d5,:b4,nil,:c5,
                    nil,:f5,nil,:f5,:f5,nil,nil,nil,
                    nil,nil,:e5,:ds5,:d5,:b4,nil,:c5,
                    nil,:e4,:f4,:g4,nil,:c4,:e4,:f4,
                    nil,nil,:gs4,nil,nil,:f4,nil,nil,
                    :e4,nil,nil,nil,nil,nil,nil,nil], [0.2])
```

Credits: https://gist.github.com/xavriley/87ef7548039d1ee301bb

The musicians are synchronized with each other, who are in turn synchronized by the conductor, who sets the tempo (locks and unlocks everyone).
Hint: consider a pattern with controllers (sync/cue)

1° thread/live loop

```
use_bpm 100
  use_synth :pulse
  use_synth_defaults release: 0.2, mod_rate: 5, amp: 0.6
play_pattern_timed([nil,nil,:e5,:ds5,:d5,:b4,nil,:c5, nil,:e4,:f4,:g4,nil,:c4,:e4,:f4,
nil,nil,:e5,:ds5,:d5,:b4,nil,:c5, nil], [0.25])
```

2° thread/live loop

```
use_synth :tri
  use_synth_defaults attack: 0, sustain: 0.1, decay: 0.1, release: 0.1, amp: 0.4
 play_pattern_timed([nil,:f5,nil,:f5,:f5,nil,nil,nil], [0.25])
```

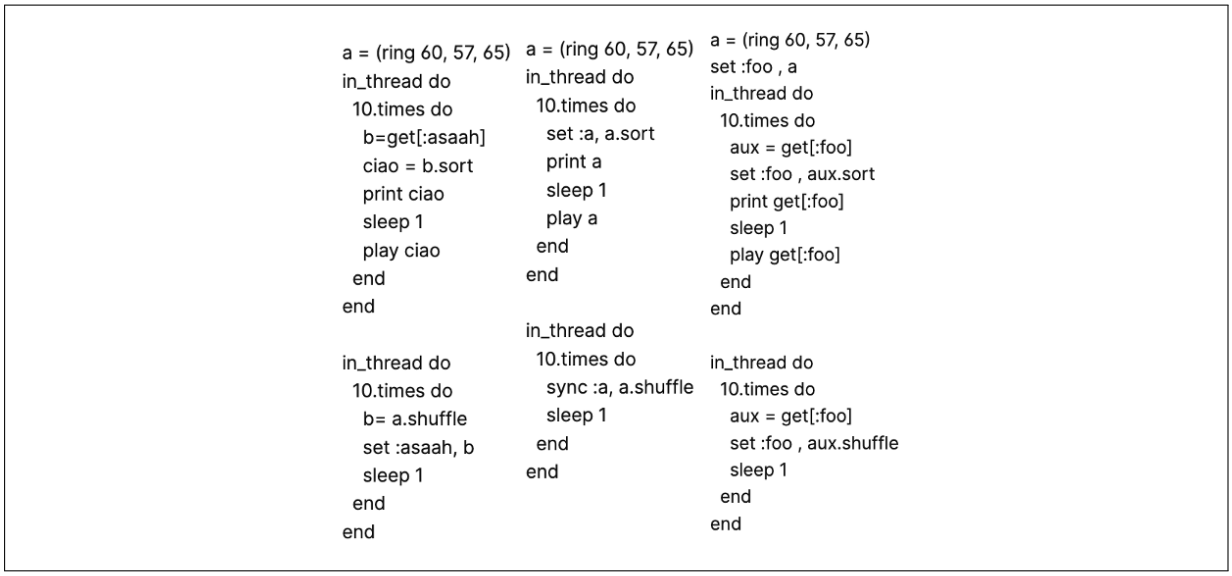
3° thread/live loop

```
use_synth :tri
  use_synth_defaults attack: 0, sustain: 0.1, decay: 0.1, release: 0.1, amp: 0.4
play_pattern_timed([nil,nil,:gs4,nil,nil,:f4,nil,nil, :e4,nil,nil,nil,nil,nil,nil,nil], [0.25])
```

4° thread/live loop - Conductor

```
# Do stuffs
```

Task 7 simulates a memory barrier, where four threads, including three musicians and a conductor, interact. The musicians synchronize to play a melody, while the conductor pauses the other threads when a counter variable reaches a certain value. After a set time, the conductor sends a message for the threads to resume playing. The task requires students to synchronize the musicians and implement the behavior of the conductor thread, responsible for setting the tempo for the other threads.



```
a = (ring 60, 57, 65)
in_thread do
 10.times do
  b=get[:asaah]
  ciao = b.sort
  print ciao
  sleep 1
  play ciao
 end
end

in_thread do
 10.times do
  b= a.shuffle
  set :asaah, b
  sleep 1
 end
end
```

```
a = (ring 60, 57, 65)
in_thread do
 10.times do
  set :a, a.sort
  print a
  sleep 1
  play a
 end
end

in_thread do
 10.times do
  sync :a, a.shuffle
  sleep 1
 end
end
```

```
a = (ring 60, 57, 65)
set :foo , a
in_thread do
 10.times do
  aux = get[:foo]
  set :foo , aux.sort
  print get[:foo]
  sleep 1
  play get[:foo]
 end
end

in_thread do
 10.times do
  aux = get[:foo]
  set :foo , aux.shuffle
  sleep 1
 end
end
```

Figure 3.2: Task 4 Solutions

3.8 Experimental Evaluation

In this section we describe the two teaching experiments realized to evaluate the approach, first presenting the setting and then the obtained results.

3.8.1 Experimental Set-Up

The first experiment involved third-year students enrolled in the CP course, while the second experiment involved first-year students enrolled in the CA course on computer architectures.

The CP course is a third-year, second-semester course that focuses specifically on the design and analysis of algorithms for concurrent and distributed systems. It has a unique position in our B.Sc. program in CS as the only course dedicated entirely to concurrent programming. Students enrolled in this course have advanced knowledge in several areas, including programming (non-concurrent), databases, networks, and architectures. This foundational knowledge serves as a basis for exploring the intricacies of concurrent programming and gaining a deeper understanding of its principles and applications.

The CA course is an annual first-year course that focuses on teaching students the fundamentals of computer architecture design for modern microprocessors. Throughout the course, students gain knowledge and skills in several areas, including assembler languages, number representation and arithmetics, combinatorial and sequential circuits, and processor and memory hierarchies.

From the 2022/2023 academic year, students have been introduced to concurrency through our experimental approach and then with a very brief introduction to multi-threading in C/C++ using the `pthread` and `std:thread` libraries. It is important to note that students enrolled in the CA course already have one semester of experience in imperative programming using C++.

The CP students are referred to as the P1 population, while the CA students are referred to as the P2 population. Specifically, P1 consisted of 54 third-year students (divided into 10 teams), while P2 consisted of 130 third-year students (divided into 34 teams).

Both experiments described in the chapter were conducted during the second semester of the 2022/2023 academic year. The first experiment took place at the beginning of the CP course, where students were introduced to concurrency concepts. The second experiment was conducted in April, just before students in the CA course were introduced to multi-threading in C. The timing of these experiments allowed students to gain a basic understanding of concurrency through the Sonic Pi-based approach, before delving into multi-threading in other languages.

The activities did not contribute to the student's final summative assessment. Instead, they had formative value, serving as learning experiences and opportunities for students to develop their understanding of concurrent programming concepts.

During the activities, we collected team APP tasks to analyze students' understanding of concurrency and any misconceptions that might arise, according to the rubric described in Appendix **??**. We also collected data through an anonymous individual post-questionnaire on students' appreciation/perception of the activity. Finally, we collected the results of a post-individual test to investigate knowledge transfer. The test consisted of three exercises formulated either in pseudocode or in the C programming language. These exercises proposed a similar concurrency scenario, but in other languages, and provided practical opportunities for students to apply their understanding of concurrency in a different learning context.

Technically, the final questionnaire and the "knowledge transfer" test were developed using Google Forms, which provides a user-friendly interface for data collection. The questionnaire and the test can be found in the appendix A.1 and A.2. The TBL quizzes were developed using the Moodle platform, which provides a comprehensive set of features for online learning and assessment. In addition, we used platforms such as Wooclap[11] and Padlet to collect the team task solutions and responses. These platforms provided a collaborative environment where teams could share and submit their solutions, encouraging student engagement and teamwork. Overall, the combination of these tools facilitated efficient data collection and supported the interactive and collaborative aspects of the learning activities.

[11] https://www.wooclap.com

3.8.2 Experimental Results

3.8.3 RAT Results

CA The questions included in the RAT quiz have been carefully selected based on a refinement process of similar past activities conducted with Sonic Pi. The test aims to assess the students' understanding of some basic concepts related to concurrency and live coding, which are essential for completing the Team App tasks in the TBL module. Due to time constraints and the structure of the TBL module, only 5 questions were chosen, each focusing on independent concepts. The data collected from the quiz were not used to answer the research questions, but are reported for completeness to present the full material of the TBL activity, as the quiz served as a preparation tool for the subsequent Team App phase. Notably, the repeated quiz within the team generally yielded better results. However, it was not our intention to statistically validate this result using psychometric or other techniques to assess validity and reliability, as these data were not used in the research study. The results are reported in Table 3.1 as additional material for completeness only. In the following section we focus our attention on the results obtained for the Team APP tasks.

As expected, we found that team scores tended to be higher than individual scores. Specifically, the RAT quiz success rate (i.e., the percentage of correct answers) increases from 5.6 for the individual test to 6.5 for the team test. Table 3.2 shows the percentage of total correct answers for each item Q_i. We found that teams outperformed individuals in all but the first question, which was about live coding loops. Specifically, Q_1, 19 teams answered correctly, and 6 students voted for "No effect on execution. This is live coding", 7 students voted for "It forks the main thread and creates two new ones. One will play 60 and the other 65. This is live coding", and 1 student voted for "It raises a runtime error". Looking at the individual responses, we can see that the second question was the most difficult, which was about live coding with live loops and a sample outside the loop.

CP For the CP results, table 3.3 shows the percentage of total correct answers for each item Q_i for both individuals and teams. It can be seen that the teams achieved significantly higher percentages of correct answers than the CA students, performing better in all five questions. For the CP students, the fourth question was the most challenging, which was about live coding with live loops and the get/set synchronization mechanism.

3.8.3.1 Team APP Results

We conducted both item and code analyses of the team tasks to identify potential errors and misconceptions. Specifically, for multiple-choice questions, we tried to identify patterns in incorrect answers to determine which concepts were common misconceptions. For tasks that required

CA	Q_1	Q_2	Q_3	Q_4	Q_5
Individual	75%	37%	49%	44%	60%
Team	59%	63%	59%	63%	81%

Table 3.2: CA individual and Team RAT results.

CP	Q_1	Q_2	Q_3	Q_4	Q_5
Individual	61%	46%	48%	37%	70%
Team	91%	91%	82%	73%	91%

Table 3.3: CP Individual and Team RAT results.

students to write code, we analyzed their solutions to identify specific errors. The results are presented per task, each time distinguishing between the CA and CP scenarios.

The results of the team APP are presented below.

Task 1 60% of the CA teams answered the first question correctly. However, 13% incorrectly voted that the threads in the first and third programs use a shared resource, resulting in a nondeterministic program. In addition, 7% incorrectly voted that there are no shared resources in any of the programs, making them deterministic. Finally, 20% voted that the first and third programs had no shared resources and were therefore not deterministic. These responses highlight misconceptions around the concept of shared resources, especially in the first exercise where the misconception is "reinforced" by a variable scope error (confusing local scope with global scope). In the second exercise, it is clear that there is a lack of clarity among these teams on the definition of a shared resource. There is also a misunderstanding about the meaning of a deterministic program, since a program can be non-deterministic even without shared resources between threads. Finally, the last answer reveals a misconception of both shared resources between threads and variable scope, where local variables with the same name are mistakenly considered to be shared between threads.

The CP of the teams answering correctly. Only 10% voted for the response that all three programs do not have shared resources.

Task 2 In task 2, 48% of CA teams answered correctly. 19% voted that in the programs where the function "foo" is defined, there cannot be any race conditions. This highlights a poor understanding of the program, as there are still data races present in the second program. There

is a misconception that, because it is pass-by-value, a race condition can never occur regardless of the context. Additionally, 5% of the teams voted that all programs have a mutual exclusion mechanism for accessing the shared resource. In this case, it is evident that there is a lack of understanding of concurrent mental models, where it is not clear what a synchronization mechanism is for protecting access to a resource. Finally, 29% voted that no answer was correct, failing to recognize the presence of a data race in the second program as well.

In this case, the results for CP are different. The percentage of correct answers is lower, standing at 30%. 70% of the teams voted that none of the answers were correct, thus failing to recognize the existence of a data race in the second program.

Task 3 In task 3, 67% of the CA teams and 78% of the CP teams answered correctly, stating that the output was the chord of A minor. However, 27% of the CA teams (and 22% of the CP teams) failed to recognize the presence of a race condition, and during the plenary discussion, the misconception emerged that a data race is necessary for a race condition to occur. Finally, 6% of the CA teams answered that the notes are always played in the same order, indicating that their mental model associates an implicit sequential order to the creation and execution of the three threads, thus not recognizing the race condition.

Task 4 Tasks from 4 onwards are exclusive to the CP experiment. Figure 3.2 displays three solutions to task 4, which encompass all the other cases. All teams recognized the problem (the thread orders the list but it is not always played in order) and applied a strategy to solve it. However, in the solution on the left, the correct synchronization objective was not identified. The solution was devised based on basic knowledge, which involved creating local copies of the shared variable but failed to eliminate the underlying race condition. In the center solution, there is a partially correct approach, but it exhibits misconceptions regarding synchronization mechanisms, as both a lock and a wait signal are utilized, albeit the wait signal is used incorrectly. Finally, the right solution demonstrates a correct approach (excluding some syntax errors) where locking mechanisms are applied to the shared resource.

Task 5 In Task 5, 50% of the teams identified the deadlock situation. 10% described it as starvation, while 30% believed that one of the two threads would eventually get all the resources. Finally, 10% answered that it was something else. In this case, the students displayed an inaccurate mental model of program execution, failing to recognize that the two threads were stuck waiting for resources held by each other, preventing any progress in the program.

Task 6 & 7 All teams successfully completed Task 6 with the correct solution. During the activity, there were some challenges related to the specific scope rules of Sonic Pi. However, once the dual scope of synthesizers and variables was clarified by us, no further issues arose. The

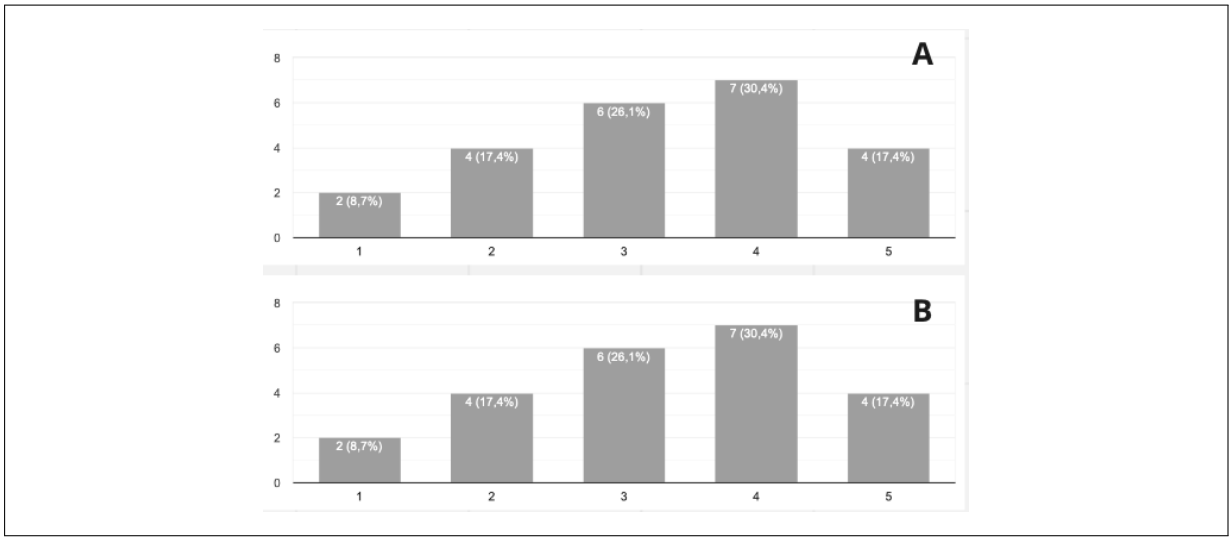


Figure 3.3: CA final questionnaire

teams effectively utilized the knowledge acquired from previous tasks. Task 7, on the other hand, proved to be more challenging as it was a less guided exercise compared to the previous one. It required a deep understanding of thread synchronization mechanisms and the management of a shared resource. Not all teams submitted a solution as they were unable to translate the problem into a practical solution. Others partially implemented a solution where the conductor coordinates the various threads but fails to pause/resume execution in a "memory barrier" style. Only one group managed to successfully implement the required solution. Team results can be found in the appendix A.3.

3.8.3.2 Final Questionnaire

CA A total of 23 people responded to the final CA questionnaire. None had any previous experience of concurrent programming and Sonic Pi or Ruby. For the question "I think the musical approach with Sonic Pi is useful for understanding concurrency" [A] the median is 4. For the question "I think the activity was effective in introducing and/or deepening some concepts of concurrent programming" [B] the median is 3. In particular, the students who gave a negative or neutral answer to question A were the same students who gave a neutral or negative answer to question B. Figure 3.3 shows the distribution of responses on a Likert scale from 1 to 5.

Participants commented that the activity was useful for understanding the importance and benefits of concurrent programming, learning a new programming language, understanding the concept of threads and synchronization, and improving their knowledge of concurrent programming.

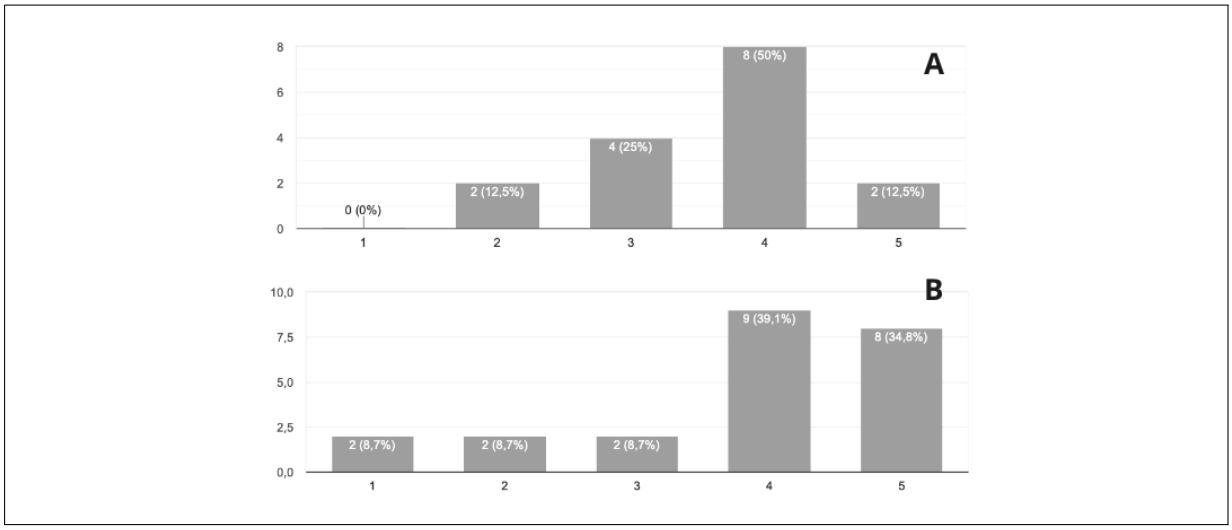


Figure 3.4: CP final questionnaire

74% of participants found the activity to be at an appropriate level of challenge, 22% found it difficult and the remaining 4% found it easy. Regarding the time allotted for the activity, 65% of participants felt it was sufficient, while 35% expressed the wish for more time to complete the tasks. In terms of individual preparation, 6 people consider that they have prepared adequately for the activity, 14 do not know, and 3 insufficiently. Finally, 22% of students found the teamwork experience very useful, 30% useful, 26% neutral, 13% not very much, and 9% not at all.

CP The final CP questionnaire was completed by 16 students. Of these, 44% had previous work experience in concurrent programming and 30% had used Sonic Pi in the ICDD course. For the question "I think the musical approach with Sonic Pi is useful for understanding concurrency" [A] the median is 4. For the question "I think the activity was effective in introducing and/or deepening some concepts of concurrent programming" [B] the median is 4. In particular, students who gave a negative or neutral response to one question were the same as those who gave a neutral or negative response to the other question. Figure 3.4 shows the distribution of responses on a Likert scale from 1 to 5. In addition, several students commented that the approach provided a useful introduction to concurrency and offered a clear practical example. In addition, 80% of the students reported that they found the level of difficulty of the activity to be fair, while 18% found it to be difficult. Regarding the time allocated, 70% of the students felt that it was fair, while the remaining students expressed a preference for more time to complete the tasks. In terms of individual preparation, 44% people consider that they have prepared adequately for the activity, 31% do not know, and 25% insufficiently. Finally, 25% of students found the teamwork experience very useful, and 75% useful.

3.8.3.3 Assessing Knowledge Transfer

CA Only 6 students answered the first exercise correctly. In particular, most of the incorrect answers revealed a mental model still based on sequential execution, where note 60 would be played only after the other thread had played notes 45 and 50, thus describing deterministic behavior. One student confused the behavior of the `cue` with that of `synth`, while two students confused `in\_thread` with `live\_loops`, stating that the sound would be repeated an infinite number of times. Finally, one student replied that only notes 45 and 60 would be played.

In the second exercise, in pseudo-code, all but one of the CA students correctly identified the possible scenarios. Only one student said that the program always generates an error.

In the third exercise, using C with the pthread library, 80% of the CA students answered that the program was not deterministic. However, only 15% of them correctly identified the possible outputs of the program. The remaining students recognised the correct outputs but also added the combination 0 0 or 0 1 10, which are impossible as they can never occur. These answers once again highlight the problem of misconceptions about basic concepts, particularly references and pointers.

CP In the CP experiment, 7 out of 15 students solved the first exercise correctly, while the others either did not answer or considered a sequential model. The second exercise was solved correctly by all students. Finally, 80% answered the third exercise correctly (non-deterministic and correct output), while the remaining students judged the behavior of the program to be non-deterministic.

3.9 Discussion

TBL Methodology An innovative aspect of our approach is the adoption of a TBL pedagogy. This method combines elements typical of flipped-class with individual and team activities in a series of focused tasks that perfectly fit our goal, i.e., exploring concurrency misconceptions and common mistakes. The strict integration of the activity with the considered courses, e.g., knowledge transfer in order to formulate problems seen in Sonic Pi in the languages adopted in the courses, feedback on the proposed quiz and task to the student involved in the course, combined with a generally positive evaluation of the activity represents a first step towards the stable application in a future edition of our bachelor degree.

A general positive evaluation of the activity represents a first step towards the stable application of TBL in future editions of our bachelor's degree.

TBL results and post-questionnaires allowed us to perform a quantitative and qualitative analyses of the outcome of our experiments. It is also important to remark that the population considered in our experiments consisted of 184 university students. Furthermore, the proposed method is the result of a refinement along three different academic years involving around 400 students in total.

The high number of students, involved voluntarily, seems to indicate the widespread perception of the usefulness of innovative approaches to programming education.

Sonic Pi for Introducing Concurrency In our opinion, there are several reasons for adopting Sonic Pi as an introductory language for concurrency. First of all, music provides a stimulating and creative domain in which to start "thinking concurrently". Furthermore, Sonic Pi is based on Ruby with advanced programming aspects such as thread-local variables and hot-swapping code, thus representing a stimulating language for university students. The sound manipulation library is interesting by itself since it provides operations to create new sounds, to modify, combine, and reproduce music samples. As in nested variable scopes, filter declarations can be nested to create a sort of effect scope that can be confined within a single thread. Concerning programming constructs, another distinguishing feature is that Sonic Pi provides constructs at different abstraction levels for creating asynchronous threads: the `in_thread` command is closer to the C `pthread_create` invocation but with a much simpler syntax; the `live_loop` embeds a built-in infinite loop and, thus, it resembles the typical `demon` pattern used in the most common system programs. Hot-swapping code makes the editor a very effective tool for experimenting with all features of the Sonic Pi programming language and, in particular, with those related to concurrency and synchronization.

In our concurrency misconception exploration, the possibility of using sound and music creation not only as the target of a given exercise but also as a concrete means to hear anomalies (e.g., data races, missing synchronization, out-of-order executions) turned out to be a key ingredient of our experiments. For instance, we exploited this feature in different tasks included in the Team App, e.g., to hear the difference between a correct program (which expected result is to reproduce a chord) and an incorrect one (which result is a random sequence of the notes in the chord).

Being a domain-specific language can be seen as a possible downside for using Sonic Pi to introduce concurrency in the first year of undergraduate courses. Introductory courses are typically based on widespread languages such as C and C++ and are mainly focused on programming methodologies and data structures. To overcome this problem, in our experiments, Sonic Pi has been introduced in the context of a mini-course together with the TBL method. Together they were also meant as innovative methods to stimulate the students' attention and learning process.

The simplified concurrency abstractions of Sonic Pi combined with a modern programming style and a domain-specific flavor is a perfect fit for first-year university students.

Data Analysis The results obtained from the tasks highlight the presence of misconceptions among the participants, both about the pure concurrent concept and about fundamental programming concepts within a concurrent scenario. For example, participants showed a lack of clarity in distinguishing between local and global scope and in understanding how shared resources affect program behavior, and some others mistakenly believed that certain programming mechanisms, such as pass-by-value, could eliminate the possibility of race conditions.

The results seem to highlight the importance of addressing misconceptions with targeted, guided interventions aimed at making concepts and scenarios about concurrent programming explicit. Furthermore, the plenary discussion generated during the activity could help participants in consolidating their understanding and develop a viable mental model. A notable finding is that addressing misconceptions is also useful for third-year students, where one might expect them to have acquired advanced programming skills.

In terms of knowledge transfer, it appears that students found it easier to align their existing mental model of programs in Sonic Pi with the pseudo-code representation. On the other hand, the exercises using C syntax required more effort in terms of adapting a mental model from one programming language to another with a different NM.

Generalizability of Results First, the tasks may not cover all aspects of concurrent programming, and the results may not fully capture students' overall understanding of concurrency. Our tasks were designed with only a few misconceptions in mind. As a result, the assessment of students' understanding is based on their responses to specific tasks and may not reflect their overall understanding of concurrent programming concepts. In addition, the sample size in both experiments may not be representative of the entire student population, which may limit the generalisability of the results, especially for the knowledge transfer part. In fact, only a small fraction of the students responded to the final questionnaire, which limits the significance of the results. Therefore, the generalisability of the knowledge transfer results should be interpreted with caution and further research is needed.

The data analysis highlights the importance of addressing misconceptions with targeted, guided interventions aimed at making concepts and scenarios about concurrent programming explicit.

3.10 Conclusions

In this chapter, we presented an introductory approach to concurrency aimed at exploring undergraduate students' misconceptions using the Sonic Pi language so as to exploit the natural connection between multi-threading and live music coding. In our view, this chapter serves as a pilot study and provides preliminary evidence for the CS education community.

More specifically, we have discussed the methodology adopted in our experiments and the population settings, the material developed for the RAT quiz and Team app activities, and commented on the data collected before, during, and after the activities. The experiment comprises a further transfer learning experiment designed via a series of exercises proposed both in Sonic Pi and more traditional multi-threaded programming languages such as C and C++.

Promising initial results from questionnaires and tasks provide strong motivation for us to delve deeper into live music coding and targeted exercises, particularly those related to program comprehension. Our results are consistent with the existing literature on misconceptions, suggesting that students may face challenges not only in grasping concepts related to concurrency but also in understanding fundamental principles underlying the computational model. These considerations also apply to third-year students, leading us to reflect on the pedagogy of programming education in our undergraduate program.

In particular, the results of the Team Apps and the plenary discussion seem to suggest that a focus on mental models is beneficial in improving concurrency education, both to teach viable mental models and to present specific cases where common non-viable mental models are inappropriate and lead to misconceptions.

The results of the knowledge transfer test are quite encouraging, at least for the pseudo-code scenario, but future research is needed.

Based on the first year's results, we believe that the proposed approach could also be valuable in secondary education, especially in high schools with a focus on STEM education or in vocational schools specializing in technology-related fields.

Finally, future experiments will provide fertile ground to further investigate the effectiveness of our approach and to analyze and address other misconceptions that are not included in the current research.

Data availability

The data that support the findings of this study are openly available on Github https://github.com/researchDataset/SonicPi_dataset/tree/main.