Universiteit Leiden
The Netherlands

Opleiding Informatica

A Concurrent

Visual Programming Language

Oualid Azzeggarh

Supervisors:
Hans-Dieter Hiep & Walter Kosters

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

13/07/2023

Abstract

This thesis explores the concept of visual programming language (VPL) design and its implications for the field of concurrency. To ease onboarding for new developers to the world of concurrency, this thesis tries to answer the question: "How can a VPL for concurrency improve the conventional way of concurrency programming?". By developing a node-based VPL, users can develop code by stringing up nodes to each other. Concurrency is implemented in the form of the fork node and mutex zones, mimicking forking and mutex locks, respectively. The user can simulate the code with various schedulers, allowing them to explore how the same piece of code behaves differently depending on the said scheduler. These features allow the user to better grasp the concept of concurrency without being bound to any language.

Contents

1 Introduction

In recent years, computers have witnessed remarkable advancements in computational power, surpassing what was once unimaginable just a few decades ago. The program design has shifted from primarily focused on optimization to harnessing its full potential [Sch97]. While single-threaded applications may suffice for many scenarios, the benefits of concurrent programming should not be overlooked, particularly as CPUs continue to incorporate more cores, thus offering opportunities for enhanced performance. However, embracing concurrency is not without its challenges, and it is crucial for new developers to grasp its concepts early on.

Concurrent programming is a challenging aspect of software development that often poses difficulties for developers [AB18]. The complexity of managing multiple threads or processes, synchronization, and communication can be overwhelming when expressed through traditional textual coding. To address these challenges and assist new developers in gaining a better understanding of concurrency, this thesis explores the potential benefits of a concurrent Visual Programming Language (VPL).

One of the primary hurdles faced by developers when learning concurrency is the complexity of managing mutex locks. Mutex locks ensure that a thread follows a specific code flow until it is unlocked, preventing data conflicts. However, when expressed textually, it can be challenging to comprehend how the code flows within a concurrent system. Additionally, concurrency introduces complications such as interleaving and race conditions, further adding to the difficulty of concurrent programming.

1.1 Objective and Scope

To address the challenges of understanding and implementing concurrent programming, we propose a Visual Programming Language (VPL) specifically designed to improve the learning experience. The VPL showcases various possibilities of program flow, utilizing mutex zones instead of mutex locks. These mutex zones clearly define sections where code must be executed exclusively by a single thread before another thread can proceed. In the VPL, variables are immutable by default, and developers can choose to make them mutable if needed. While this VPL is a proof of concept, it serves as a valuable scratchpad for developers to prototype concurrent functions before integrating them into their projects. The central research question guiding this thesis is:

> *How can a Visual Programming Language for concurrency improve the conventional way of programming concurrent systems?*

This thesis aims to examine the advantages that a VPL brings to the domain of concurrent programming. By presenting a visual representation of concurrent processes and their interactions, a VPL offers an alternative approach that simplifies complexity, reduces errors, promotes intuitive design, facilitates learning, enables rapid prototyping and iteration, and fosters collaboration among developers. While this thesis does not explore or develop a VPL capable of compiling production-level code, it aims to assist new developers to understand concurrent programming. It aims to provide a comprehensive evaluation of the benefits of a concurrent VPL, empowering new developers to leverage this approach and contribute to the advancement of concurrent programming practices.

1.2 Structure

This thesis is structured as follows. Section 2 lays down the literature and background used. Section 3 defines the VPL and how it contrasts with textual programming in various scenarios. Section 4 describes the implementation and technical aspects of the VPL. Second to last, Section 5 discusses the benefits and drawbacks of the VPL, and as well as describes the limitations and what can be done if this research is developed further. Lastly, Section 6 provides concluding thoughts on the research done.

This thesis is conducted at Leiden University, supervised under Dhr. Hiep and Dhr. Kosters.

2 Background

This section will first describe the concepts for the design of a VPL and their efficacy in the learning process. The following subsection provides an overview of concurrency. This helps with understanding the modes of communication section, which describes how concurrency is implemented in programming languages. A brief background of the JavaScript frameworks is also provided after the concurrent programming sections before ending this section with the existing work related to this research.

2.1 Visual Programming Languages (VPLs)

This subsection defines what a VPL is, how it can be represented, and how it is used to learn programming concepts.

2.1.1 Defining a VPL

A VPL utilizes graphical elements to construct a program. This distinct difference from the traditional textual programs allows VPL designers to design their VPL in two ways: Block-Based and Node Based.

Block-based VPLs Block-based VPLs allow users to create programs by stacking code blocks together to form functions or procedures. These VPLs aim to provide beginners with an intuitive way to understand programming concepts without being overwhelmed by syntax complexities. Scratch [RMMH+09] is a popular example of a block-based VPL, designed specifically for children to create their own games and animations. See Figure 1.

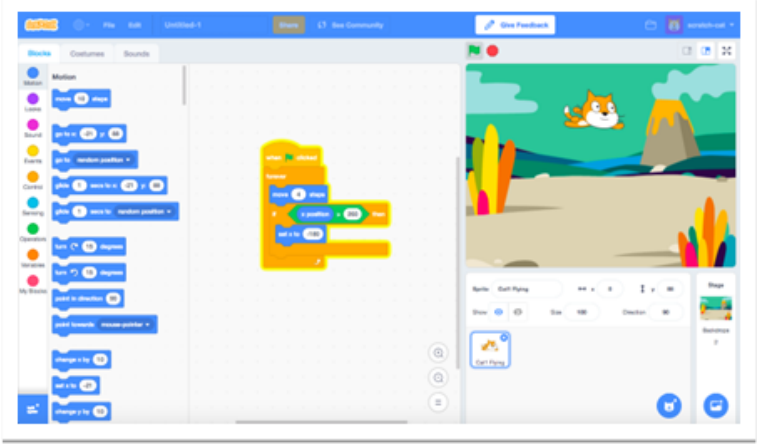

Figure 1: Scratch's Block-Based VPL. Source: https://scratch.mit.edu/

Node-based VPLs Node-based VPLs enable users to create programs by connecting nodes, similar to Petri-Nets (further explained in Section 2.3.1). Typically, a program starts from a start node and sequentially executes connected nodes. Some node-based VPLs allow users to create their own custom nodes by grouping a set of nodes together, enhancing modularity and clarity. Notable examples of node-based VPLs include Blender [Ble], a 3D modeling package, and Unreal Engine 5

[Unr], a 3D real-time creation tool. Unreal Engine's Blueprint VPL enables artists and developers alike to develop their vision in real-time at high efficiency.

Figure 2: Unreal Engine's Node-Based Blueprint VPL. Source: https://docs.unrealengine.com/5.2/en-US/blueprints-visual-scripting-in-unreal-engine/

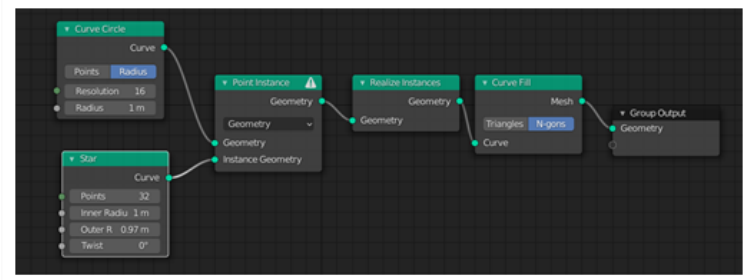

Figure 3: Blender's Node-Based Geometry Node VPL

Shown above in Figure 3. Blender's geometry nodes[Ble] enable artists to programmatically create 3D models in an intuitive manner by visualizing how the modifications on geometry are being applied. An artist can quickly create procedurally generated models in a fraction of the time compared to the normal way.

2.1.2 Learning Programming Concepts

The aforementioned programs are able to extend the programming concepts from developers to artists, game designers, and even young children interested in making their own games. This is not just because the end result is to make a game or an art piece. Previous research [Tsa18] has shown that the test subjects, students, learning abstract programming concepts through VPL outperform those that do it through textual programming. Frustration is reduced through the instant feedback design-based learning provides and thus, students are able to reflect, correct, and understand the relationship between the concepts learned and the project they are designing.

The research indicated an increase in the performance of students with low to moderate self-efficacy. As the user-friendly design of the VPL lowered the barrier of the compilation process, these students were able to quickly ease themselves into the learning process [LK14]. By sidestepping syntax and other setup work required to initialize the project, a broader audience can be captured.

2.1.3 Implications for this research

For this research, a VPL will be constructed containing concurrent programming functionality. As concurrent programming by its nature leads to handling multiple threads, a node-based solution would seem to be more suitable than a block-based solution.
The aim would be to encourage students to play with the concept of concurrent programming by selecting different schedulers and understanding the different behaviors it can cause. As the VPL provides instant visual feedback, the user can understand faster how different schedulers can affect the code.

2.2 Overview of Concurrency

This subsection provides an overview of concurrency by providing findings regarding concurrency models, communication methods, and concurrency modules. By understanding these concepts a simplified version can be designed for the VPL.

2.2.1 Definition

In essence, concurrency just means that multiple computations are happening at the same time.

2.2.2 Threads

Threads are lightweight units of execution within a process. They share the same memory space and can communicate with each other directly. Threads allow for concurrent execution within a single program, dividing and executing tasks simultaneously. They are commonly used in multi-threaded programming to achieve parallelism and exploit multiple CPU cores. Threads can share data efficiently but require careful synchronization to avoid data races and ensure proper coordination between threads.

2.2.3 Communication

Communication is necessary in order for the independent threads to behave as intended. Communication within the field of concurrent programming can be divided into two categories: hidden and explicit. As this thesis is focused on providing users with a VPL whereby they can learn concurrency intuitively, concepts about how a hidden communication is implemented have not been explored. More information regarding explicit communication can be found in Section 2.3.

2.2.4 Race condition

Coined by David A. Huffman in his thesis [Nel55], a race condition, also known as *racing*, is a scenario whereby two or more threads execute different code pieces at the same time. If the amount of time for the threads to be finished is different from what is expected, unintended behavior can occur.
Data Racing is a type of race condition whereby two or more threads access the same memory location at the same time. If one thread is intended to read the memory location before the other, but in reality the opposite happens, the result will likely differ. This non-deterministic behavior is difficult to debug and it is thus, therefore, better to avoid race conditions if possible.

2.3 Modes of Communication

In the case of explicit communication, there are two types:

Message Passing Message passing is a mechanism where tasks or processes communicate by sending and receiving messages. In this model, processes interact by explicitly exchanging messages containing data and control information. Message passing allows for loosely coupled and scalable systems, as processes can communicate with each other without directly accessing each other's memory. It promotes modularity and encapsulation, as processes are isolated and can communicate only through well-defined message interfaces.

Shared Memory Shared memory is a concurrency model where multiple tasks or processes concurrently access and manipulate shared data. In this model, a region of memory is shared among different tasks or processes, allowing them to communicate and synchronize through the shared memory. However, synchronization mechanisms such as locks, semaphores, or atomic operations need to be employed to ensure correct and synchronized access to shared data. These mechanisms help prevent race conditions and ensure that only one task or process accesses the shared data at a time. Shared memory can provide efficient communication and data sharing but requires careful synchronization to avoid data inconsistencies or conflicts.

Despite Message Passing and Shared Memory being two modes of communication, it is possible to implement one mode on top of the other. For example, message passing can be built from shared memory by using a First-In-First-Out (FIFO) channel in shared memory. Threads can write messages to be sent and read messages that have been received. Utilizing mutexes or other synchronization mechanisms, multiple threads can access the FIFO channel without conflicts occurring. It is mainly to comfortably use the strengths depending on the purpose of the program. Message-passing communication is used in distributed systems as threads run on separate machines. Shared memory is often used in multiprocessors as it allows for an efficient and low latent way of communicating data.

2.3.1 Concurrency Models

Concurrency models encompass a set of principles, mechanisms, and abstractions that govern concurrent execution in computing systems. They facilitate efficient resource utilization, improved system performance, and responsiveness. Concurrency models play a vital role in modern computing by allowing multiple tasks or processes to make progress simultaneously.

1. **Petri Nets:** Petri Nets [Rei13] were an attempt to codify rules of concurrency. The theory of data flow built upon it and led it to be used as a graphical model for modeling and analyzing concurrent systems. They consist of places, transitions, and arcs, representing states, events, and dependencies, respectively.

2. **Calculus of Communicating Systems (CCS):** CCS [Mil80] is a process calculus used for describing and analyzing the behavior of concurrent systems. It provides a formal language for specifying the interactions between concurrent processes through communication and synchronization primitives. CCS enables reasoning about process composition, communication, and process equivalences.

3. **Communicating Sequential Processes (CSP):** [Ros97] is a process algebra used for modeling and reasoning about concurrent systems. It emphasizes the composition of processes through communication channels. CSP provides operators for process synchronization, parallel composition, and communication, allowing for the specification and verification of concurrent systems.

4. **π-calculus:** The π-calculus [Mil99] is a process calculus that extends traditional process algebras by incorporating name passing as a fundamental primitive. It enables the description of dynamic process creation and communication patterns, making it suitable for modeling distributed systems and mobile computing. π-calculus supports the specification and analysis of concurrent and distributed systems.

Note that a consistency model is required for concurrency. A consistency model promises consistent results of reading and writing, and predictable updates of memory, as long as the contract specified between the programmer and system is being followed.

2.3.2 Programming with Concurrency

Depending on the model, communication can be handled implicitly or explicitly. When handled implicitly, the concurrent aspects are hidden from the programmer. The Promise feature from ECMAScript 6 [ES6] does this.

2.3.3 Implication for this VPL

As the target audience for the VPL is students and new developers, a simple thread-based shared-memory concurrency model will be designed.

2.4 Schedulers

In concurrent programming, a scheduler is a component or algorithm responsible for managing the execution of multiple tasks or threads within a program. Its primary role is to determine the order and timing of task execution, allocate system resources, and ensure that tasks progress in a coordinated and efficient manner.
Schedulers are commonly used in multi-threaded or multi-process environments, where multiple tasks or threads can run concurrently, sharing the available computing resources such as CPU time, memory, and I/O devices. The scheduler acts as a mediator between the tasks and the underlying system, making decisions on when to start, pause, resume, or terminate tasks.

The specific behavior and policies of a scheduler can vary depending on the programming language, operating system, or framework being used. Here are some of the common types of schedulers:

1. Preemptive Scheduler: This type of scheduler can interrupt a running task at any time to allocate the CPU to another task with higher priority. It ensures fairness and responsiveness but may incur some overhead due to frequent context switching.

2. Cooperative Scheduler: In contrast to the preemptive scheduler, a cooperative scheduler relies on tasks voluntarily yielding the CPU to allow other tasks to execute. This type of scheduler requires tasks to explicitly yield control, and if a task becomes unresponsive or runs indefinitely, it can affect the entire system.

3. Work-Stealing Scheduler: This scheduler is typically used in environments with a shared work queue, such as the task parallelism model. Each thread or worker has its own local queue, and when a worker finishes its own tasks, it can steal tasks from other workers' queues to maintain load balance.

The following is a detailed overview of schedulers.

Round-Robin scheduling is a simple and widely used scheduling algorithm in operating systems and task management systems. It is a preemptive scheduling algorithm that allocates CPU time to threads in a cyclic manner, where each gets equal time of the CPU's processing time.

The basic principle of a round-robin scheduler is to maintain a ready queue containing the threads ready to run. Each task is given a fixed time to execute. When a task's time expires, it is preempted, and the next task in the ready queue is given a chance to run.

The round-robin scheduler provides fairness by ensuring that each task receives an equal amount of CPU time. It prevents a single long-running task from monopolizing the CPU and starving other tasks. The simplicity of this scheduling algorithm makes it easy to implement and understand.

However, round-robin scheduling may not be suitable in all cases. As it does not consider the priority or resource requirements of the threads, threads with different execution times may not be efficiently scheduled. Short tasks may have to wait for their turn even if the CPU is available, while long tasks may suffer from frequent preemptions.

Run to Completion scheduling is a preemptive scheduling algorithm whereby processes are allocated in a specific order, with the first thread to start also being the first to complete, followed by subsequent threads in the same order.

The fundamental principle of the run to completion scheduler involves maintaining a ready queue that contains all the threads ready to execute. However, unlike the round-robin scheduler, this algorithm allows a thread to finish its tasks entirely before the next thread in the queue is called. Consequently, longer tasks may cause shorter tasks to wait, as the scheduler prioritizes completion over equal time allocation.

The run to completion scheduler provides fairness by ensuring a deterministic order of completion. It is, in essence, a linear program where threads are executed sequentially based on the initial order.

However, it is important to note that this scheduling algorithm might not be suitable for all scenarios, as it may not optimize performance efficiently, particularly in situations involving varying task lengths and specific system requirements.

Lottery Based scheduling is a probabilistic, preemptive scheduling algorithm that allocates threads based on a lottery-like mechanism. Instead of following a fixed order or a random selection, the scheduler assigns tickets to threads, and a lottery is conducted to determine which process will be executed next. The more tickets a thread has, the higher its probability of being selected in the lottery.

The key idea behind lottery scheduling is that threads with more tickets have a greater chance of winning the lottery and being allocated CPU time. However, the lottery is conducted fairly, ensuring that all threads have at least a minimal chance of being selected. This approach allows for an element of randomness while still maintaining a level of fairness in the scheduling process.

One advantage of Lottery Scheduling is its flexibility in handling priority and resource requirements. By allocating a different number of tickets to threads based on their priority or resource needs, the scheduler can give higher-priority tasks more chances to win the lottery. This enables the system to allocate CPU time according to specific requirements or constraints.

However, Lottery Scheduling may suffer from the potential for imbalances. Threads with more tickets have a higher likelihood of being selected, which means they can dominate the CPU time, leading to potential starvation for threads with fewer tickets.

Priority Based scheduling is a commonly used scheduling algorithm in operating systems and task management systems. It assigns priorities to tasks or threads and allocates CPU time based on these priorities.

In a priority-based scheduler, each task is assigned a priority value, indicating its relative importance or urgency. The priority value can be predefined or dynamically determined based on factors such as task characteristics, system requirements, or user input. Higher-priority tasks are given precedence over lower-priority tasks in CPU allocation.

The basic principle of a priority-based scheduler involves maintaining a ready queue that contains all the threads ready to run, ordered by their priority values. The scheduler selects the thread with the highest priority from the ready queue and allocates CPU time to it. If multiple tasks share the same highest priority, an additional scheduling algorithm (round-robin, lottery based, etc.) can be used to decide which thread to select.

The priority-based scheduler allows for efficient resource utilization and responsiveness by focusing on executing higher-priority tasks promptly. It ensures that critical or time-sensitive

tasks are given the necessary CPU time to meet their deadlines or requirements. Lower-priority tasks are executed only when no higher-priority tasks are available for execution.

However, it is important to consider potential issues with priority inversion or starvation. Priority inversion occurs when a lower-priority task holds a shared resource needed by a higher-priority task, leading to delays in execution. Starvation can occur when lower-priority tasks are continuously neglected or receive insufficient CPU time due to the dominance of higher-priority tasks.

Distance Based scheduling is a scheduling algorithm that takes into account the threads remaining work or distance of tasks to determine their execution order. This approach focuses on prioritizing tasks based on the progress they have made toward completion.

In a distance-based scheduler, each task is assessed based on the remaining work required to finish it. The scheduler maintains a ready queue containing all the threads, and it selects the task with the least remaining work or distance to execute next. This means that tasks closer to completion are given priority over those that have more work remaining.

By considering the remaining work, the distance-based scheduler aims to optimize resource utilization and minimize overall execution time. It ensures that tasks nearing completion are allocated CPU time to expedite their progress and potentially improve system efficiency. This approach can be particularly beneficial in scenarios where tasks have varying sizes or complexities.

The distance-based scheduler continually evaluates the remaining work of tasks and dynamically adjusts the execution order. As tasks make progress and their remaining work decreases, their priority may increase, leading to a more efficient allocation of resources.

However, it is important to note that this algorithm does not take into account other factors such as task dependencies, priority levels, or resource constraints.

2.5 Fairness in Schedulers

When considering the fairness of schedulers, it is important to evaluate how tasks or processes are allocated CPU time and whether all tasks have an equal opportunity to execute. Fairness can be assessed based on the following criteria:

1. Equal Time Allocation: Schedulers like round-robin and random aim to provide equal time slices or opportunities to all tasks, ensuring fairness in resource allocation. Each task gets a fair share of CPU time, preventing any single task from monopolizing the system.

2. Priority Consideration: Priority-based schedulers allocate CPU time based on task priorities. They aim to give higher-priority tasks more opportunities to execute, which is crucial for time-sensitive or critical tasks. However, fairness in priority-based scheduling can be challenging to achieve, as lower-priority tasks may experience delays or starvation.

3. Progress-based Fairness: Schedulers that consider the remaining work or distance of tasks, such as the Distance-based scheduler, prioritize tasks based on their progress. This approach ensures that tasks closer to completion receive more CPU time, potentially improving overall system efficiency. However, it may lead to imbalances if tasks with more remaining work consistently receive less CPU time.

The choice of a scheduler depends on the specific requirements of the system and the trade-offs between fairness, performance, and responsiveness. It is essential to carefully analyze the characteristics of tasks, system constraints, and user expectations to determine the most suitable scheduler for a given scenario.

2.6 Frameworks

The accessibility of the VPL is vital, necessitating its availability as both a web application and a stand-alone desktop application. Initially, the VPL was being developed using Vulkan [Vul] and ImGui [ImG]. However, this approach would have shifted the focus solely towards the visual aspects of the project, diverting attention from the primary goal of supporting concurrency.
To address this, the VPL now utilizes the Tauri [Tau] framework, which allows for the creation of both desktop and web applications. Tauri is built on Rust, a programming language known for its safety and performance, and offers the flexibility to integrate various JavaScript frameworks. Leveraging web development experience, the visual side of the VPL can be developed similarly to creating a website.
The Sveltekit [Sve] framework is employed for this project, which facilitates reactivity between HTML elements and JavaScript code, resulting in clean and concise component-based code.

2.7 Related Work

Development in VPLs with concurrent programming as a focus have been done before. Here are some of the related paper;

2.7.1 A bit-level concurrent visual programming language (A-BITS) and a base computation model (APC) for its development

A-BITS [AT05] is specifically designed for asynchronous and concurrent programming, with a focus on bit-level operations commonly used in fields such as video game design, multimedia, and graphical user interfaces. This language harnesses the power of bit-level concurrent computation and is built upon the Ajiro Program Circuit (APC), a concurrent model of computation. The APC model employs primitives as processes and carriers for inter-process communication.

2.7.2 BlocklyPar: from sequential to parallel with block-based visual programming

This project [SC21] introduces a set of three tutorial games designed to educate first-year Computer Science students on parallel programming concepts. The tutorial games employ a block-based VPL, providing an engaging and interactive learning experience. The challenges presented in the games are derived from students' day-to-day tasks, making them relatable and motivating for the target audience. Through the inclusion of animation components and three new programming blocks, the tutorial games effectively enhance students' understanding of parallelism and encourage them to explore parallel solutions in modern computing environments. Preliminary tests conducted with Computer Science students have confirmed the educational value of BlocklyPar as an extension of sequential programming practice.

2.7.3 Parallel programming with VPE: a case study of an integrated visual programming environment

This research[TCT97] is set in the context of parallel computing, by using a VPL to simplify the development of parallel applications. Techniques for hierarchically constructing parallel programs have been explored, presenting a practical approach using a visual interface. Additionally, an evaluation of the performance of the generated code has been conducted for a specific application, namely matrix multiplication. The results demonstrate the potential of visual technologies in aiding the multi-dimensional tasks of parallel programming, offering a promising avenue for future development.

3 The Concurrent Visual Programming Language (CVPL)

This section showcases many example programs focused on the features of the CVPL. Starting from the basics, this section continuously adds more complexity by adding control flow, variables, and finally concurrency.

3.1 Structure

In textual programming, the code goes from text line to text line. In CVPL, the code will go from the start node to the last connected node. Intersecting nodes that may modify variables, execute functions or alter the path the code follows. This section provides an in-depth overview of the anatomy of this visual programming language and how the user could reason with it.

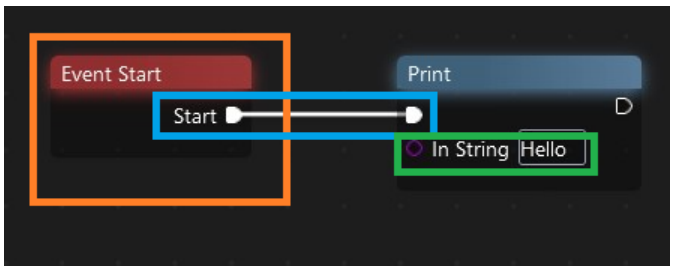

Figure 4: Basic structure of the CVPL.

3.1.1 Token

A token indicates the current place of the thread. It executes a function and/or modifies a variable when visiting the current node. The token will automatically disappear once it finishes execution on the last connected node.

3.1.2 Node

A node (shown in figure 4 indicated in orange) is an object that, when a token visits it, performs a function and then directs the token to the next node. A node can have different types: Start, Function, and Control. Only the starting node can have the type Start, which generates a token and starts the code execution. Function nodes perform actions when the token visits the node. They may use the input pins as parameters. Control nodes are used to direct the token to a different path if a condition is satisfied.

3.1.3 Parameters

The parameter pins (indicated in green) are the parameters a node takes to perform a function. These parameters can have different types based on the needs of the function. They also enable data to be transferred from node to node.

Constants are represented by an input box on the right side of the parameter pin. The user can modify the value within the input box before running the code.

3.1.4 Edge

An edge (indicated in blue) lets nodes communicate with each other. The most important line is the white execution line. It allows the token to move and execute from node to node.

3.2 Hello World

When learning any programming language, the first example of a program the documentation shows to the user is the "Hello World Program". The purpose of this program is to orient the user to the environment, syntax, and other conventions that the programming language requires in order to run the program.

```c
#include <stdio.h>

int main() {
    printf("Hello, World!");
    return 0;
}
```

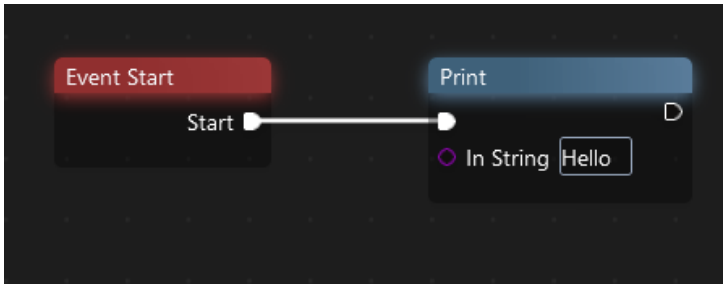

Figure 5: Code Comparison between C and CVPL: Hello World

The previous section explains the anatomy of a node-based language programming language. The syntax is from the start node, connecting nodes to get the desired program. See Figure 5.

Actions are performed whenever a node gets visited by a token. For Hello World, the only action required is to print the string "Hello World" to our output. This can be done by the Print Node.

The Print Node is a simple function node that prints out a given string input to the console log window. This string input can be either a constant written by the user or fed by a different node by means of connecting the string pin.

Going through the program semantically, it is clear that without the need for syntax or another additional setup, a user can quickly get this done. The user has effectively developed their first program by connecting the start node with the print node.
The VPL reduces the barrier of entry for non-technical users to get into coding. Due to the quickness of developing simple programs such as Hello World, the user will be able to be more confident in developing more complex programs.

3.3 Becoming Turing Complete

While being able to print out sentences the user inputs is a good way to show how to get to work in CVPL, it does not allow for any dynamic interactions. By introducing variables, control flows and arithmetic, the user can develop any kind of program within the visual programming language. To have a programming language to be Turing complete, it needs to have the following two features;

1. A form of conditional repetition or conditional jump (e.g., while, if, and goto).
2. A way to read and write some form of storage (e.g., variables, tape).

When these conditions are satisfied, the programming language can essentially create any mathematical equation/program given enough memory and or time. This subsection specifies how these conditions are met in this visual programming language.

3.3.1 Variables

With variables, problems can be generalized in a function that can give different outputs depending on the variable. Let us expand the Hello World Program by having a string variable made for it. This string variable `s` will have a default value of "Hello World".

Getters and Setters let the user access and modify the declared variables. The program can now be expanded as follows;
The Get(`s`) node reads the value in `s` and sends it to the following node to read, which is the Print Node. The Print Node reads the input value based on s and then prints that to the console log. The program is now expanded to take any string value s and outputs it to the console log.

Suppose that after the first print node, the user would like to re-use the variable to explain something else. By using the Set(`s`) node, the user is able to set a new value.
In this example, the program is further expanded to print goodbye to the user after it greeted the user. The set node conveniently has an output pin that lets the following node(s) read the variable. Essentially it is both a set and get node. This is done to reduce the number of nodes.

The user now has the ability to create, read and modify variables to simplify the programming. However, it is apparent that these variables make it more convenient for storing and loading values, the program will always behave the same way. Without conditions, the program is essentially a more hidden hard-coded program.

In the following section, the control flow of the language will be explored.

3.3.2 Conditions

When traveling, a person will take into account many variables when deciding which method of transport to take. To simplify, let us assume a person wants to reach Station Schiphol Airport from Rotterdam Central Station. They have two options: Take the Intercity or take the Intercity Direct. The Intercity Direct is a special train that will go directly to the goal, while the regular Intercity does not. For the person to be allowed to take the Intercity Direct, they would need to pay an additional fee. See Figure 6 for the code implementation in both C and CVPL.

```
...
if(hasPaid){
    TrainToBoard = "IC Direct";
}
else{
    TrainToBoard = "Intercity";
}
...
```

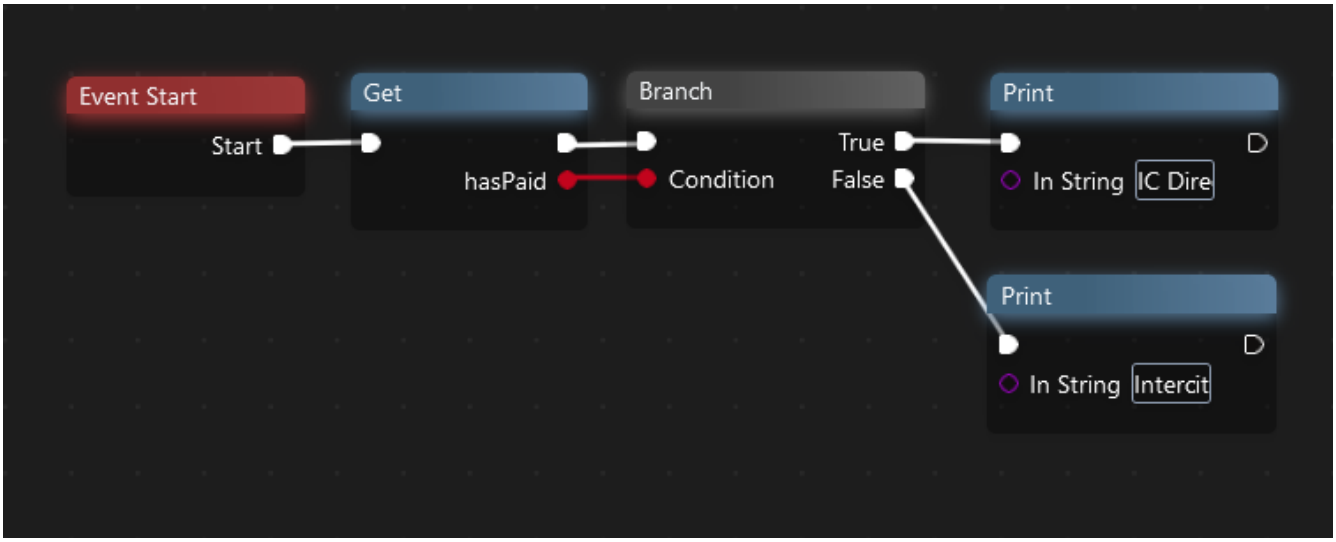

Figure 6: Code Comparison between C and CVPL: Simple if statement

It is clear that the program handles two different tasks depending on whether the person paid for the additional fee or not.

While textually it is easy to follow what is going on, it becomes more and more difficult to read the program when more nested conditions come into play.
Fortunately, in visual programming, diverting in different paths is still easy and clear to understand, allowing the user to focus on creating programs at a higher velocity. The Branch node, given a Boolean input, will divert the token to two different paths the token can take.

3.3.3 Loops

A loop is a condition whereby a piece of code will keep being performed until a condition is satisfied. See Figure 7 for how it is done in both C and CVPL.

```c
#include <stdio.h>

int main() {
    int n = 10;

    for (int i = 1; i <= n; i++) {
        printf("%d ", i);
    }

    return 0;
}
```

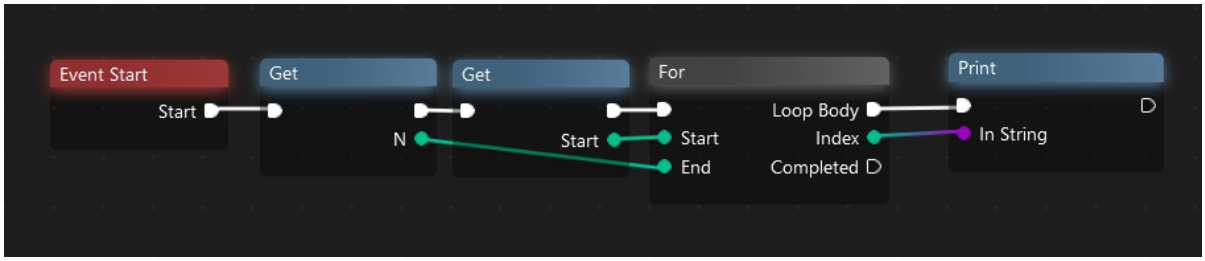

Figure 7: Code Comparison between C and CVPL: Count to N.

3.3.4 Conclusion

In this subsection, variables, conditions, and loops have been introduced. These features expanded the visual programming languages to be Turing complete. While both the textual and the visual versions are equally clear in the aforementioned examples, the intuition the visual programming language provides to new developers is to clearly show how a token flows in a program. Simple programs in visual programming languages can become verbose, but when going into harder and more complex programs, a new developer will appreciate seeing directly what path to take to.

While with just this knowledge it is possible for the user of the visual programming language to develop any program they desire, the following section will show the core benefit of this programming language in particular: concurrent programming.

3.4 Concurrent Programming

Concurrent programming in essence is doing multiple things at the same time. For a developer, implementing concurrency can be difficult as it requires careful handling of mutex locks and ensuring that race conditions will not occur. This section goes through how CVPL is able to handle it. To simplify the explanation for the examples in this section, assume that the round-robin scheduler is used and that Current Thread always goes first.

3.4.1 Fork

As shown in Figure 8, The fork node creates a new thread in CVPL starting from the New Thread Exec. pin. The user can dictate how the tokens flow depending on the way they draw the connection lines. This is really useful as it will be easy for the user to understand where the tokens originate and end. Compared to textual programming where users need to mentally picture how everything works.

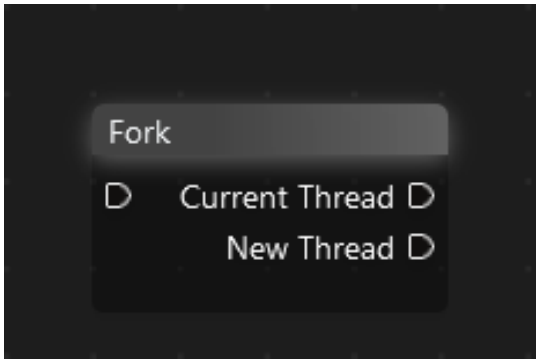

Figure 8: The Fork node in CVPL.

Figure 9 shows a basic example of how the fork node can be used.

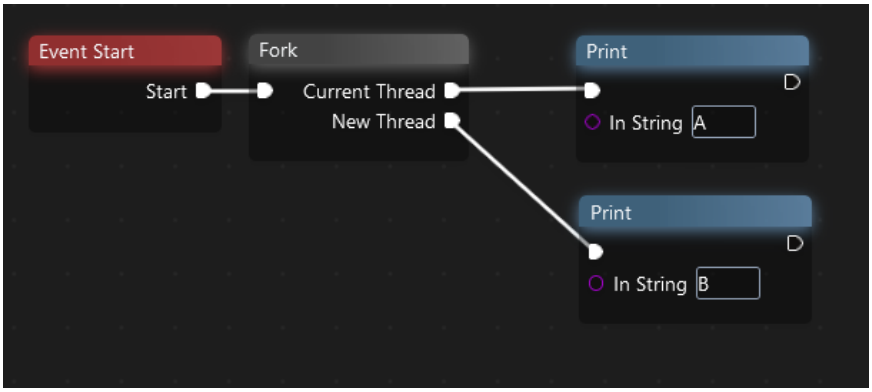

Figure 9: Basic usage of the Fork node.

The question is, however, which will be printed first? Intuitively, it would make sense that the Current Thread has priority, and thus `A` will be printed before `B`. But the Fork node does not dictate which thread has priority. Instead, the scheduler decides which thread will go first after executing the Fork node.

For the following example, assume that the Round-Robin scheduler is used. See Figure 10.

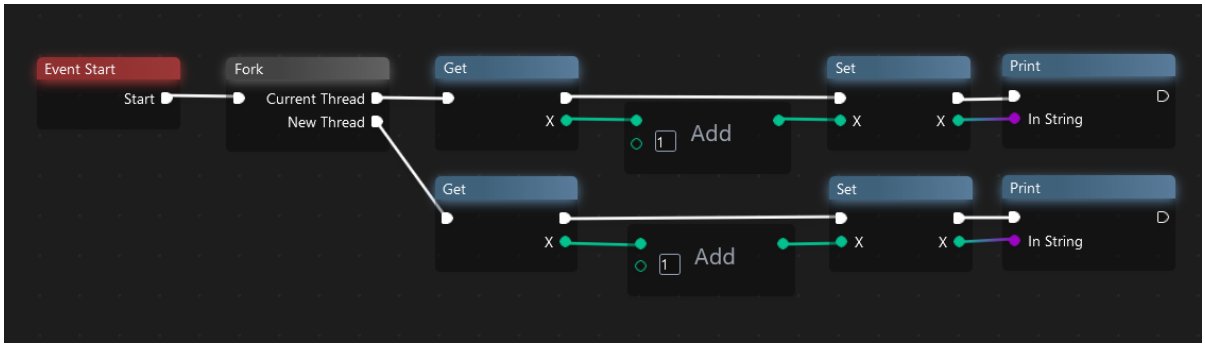

Figure 10: The goal of this code is to print X+1 twice.

The result of this code is "2" "2". This is because the threads take turns after executing each node. Thus set X is done twice before the current thread can print. In order to print "1" "2", a system must be set in place that forces priority on either of the threads.

This will be done through the mutex zone.

3.4.2 Mutex zone

Shown in Figure 11, A Mutex zone is a user-defined zone whereby the thread is forced to complete the nodes in order until it is out of the zone. This is necessary in case the user intends to specifically not want concurrency to happen in a section that handles, for example, the previous example. See Figure 12.

Figure 11: The mutex zone.

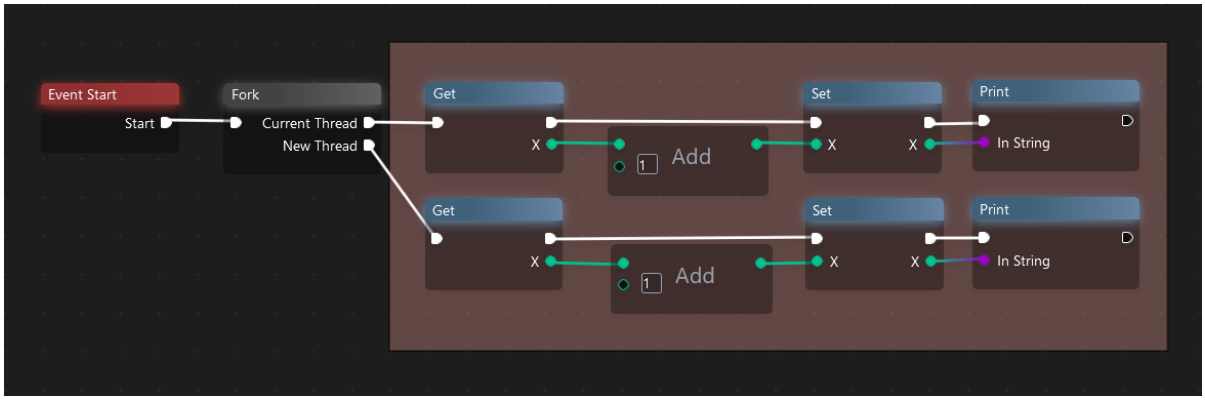

Figure 12: The goal of this code is to print X+1 twice with Mutex zone.

With the mutex zone, the code now gives the intended result of “1” “2”.

Figure 13: Wrong mutex usage. New thread is not locked out of executing.

In Figure 13, shown above, the program contains an ineffective mutex zone. The mutex zone prevents other threads from entering the zone. This does not mean that other threads are paused until the entered thread completes its journey within the zone.

The implications of this are that a user can develop their code with multiple threads intuitively by using the fork node for creating threads and the mutex zone to block-off sections. It is possible to have multiple mutex zones within each other. This is useful when the parent mutex zone has a fork node within.

3.4.3 Join

Once the user decides to continue with just one thread, it is important to clean up the unused threads. Figure 14, Shown below, is the Join Node.

Figure 14: The Join node in CVPL.

Shown in Figure 15, This node lets the user decide which thread will stay and which threads will be discarded.

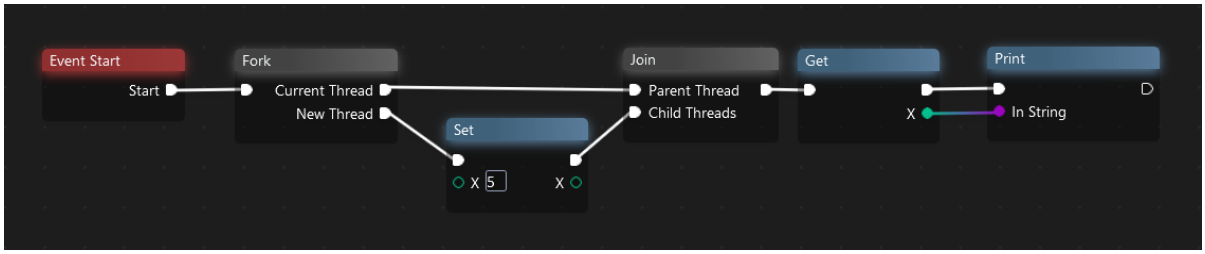

Figure 15: Usage of the Join node.

In this code example, a new thread is created that sets X as 5. The current thread does nothing but waits as the parent thread in the Join node. Once the New thread visits the Join node as a child thread, the program continues to print X as "5".

3.4.4 Merged thread lines

In CVPL it is possible for the user to connect threads to the same node. See Figure 16.

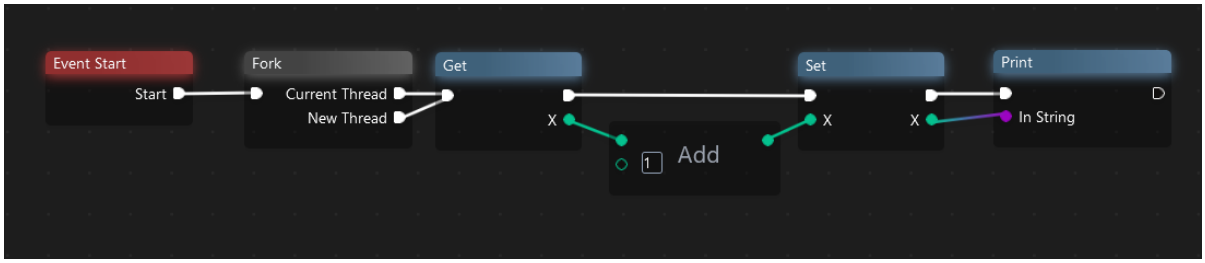

Figure 16: Merged version of previous code example.

This allows for cleaner graphs that essentially mean the same. Thus Figure 16 and Figure 17 are identical.

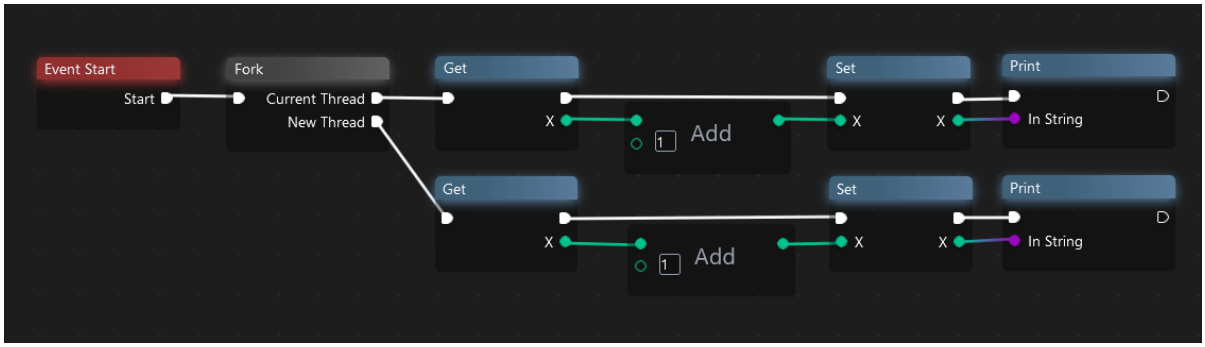

Figure 17: The goal of this code is to print X+1 twice.

Do keep in mind that while there is only one main edge after the print node, there are still threads sharing the edge.

By using a Join Node the threads can be simplified. See Figure 18.

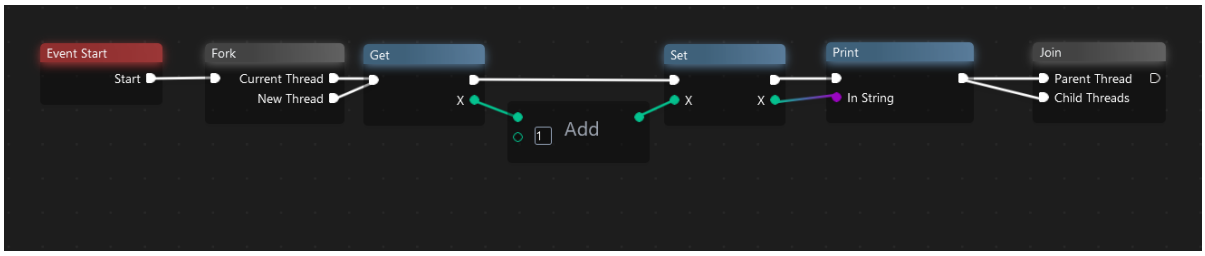

Figure 18: Merged version of previous code example.

As shown in Figure 17, there are two edges coming out from the Print node to the Join node. This is to clearly show someone viewing the code that there were multiple threads within this code path. Technically, drawing an edge only to the parent node suffices, as the Join node will set all other threads coming from the parent thread as child threads that need to be removed.
Another convenience for merging similar code paths is preventing errors with mutex zones. The following two figures (19 & 20) are identical.

Figure 19: Merged version of previous code example.

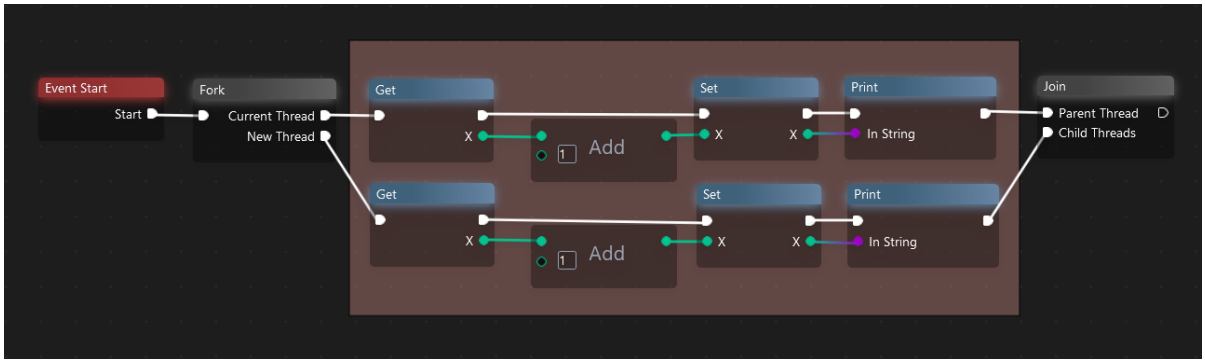

Figure 20: Merged version of previous code example.

3.4.5 Conclusion

Concurrent Programming in a visual programming language benefits the user by reducing the cognitive load when developing concurrent programs. By using visual cues like connection lines and token icons, the user has a clear idea of how to get the desired result in an efficient manner. The use of mutex zones further benefits the user in utilizing the visual programming language for critical work.
Once the program is developed, it is important that the user can also run it. As this research only simulates the programming languages, there are a handful of schedulers the user can pick from to understand how their program gets executed in different scenarios.

4 Implementation

This section explores the design and technical implementation of the developed proof of concept. Firstly, the user interface (UI) will be described, followed by the components used within the proof of concept. Lastly, the simulation options will be explained.

The proof of concept uses Tauri as it allows for the program to be used both as desktop and web application. The proof of concept has been inspired by both Unreal Engine's Blueprint VPL and Blender Geometry Node VPL.

4.1 User Interface

When developing a UI for a node-based visual programming language, it is of importance that the tools do not take attention away from the actual coding. This means that the tools should be easily accessible while not taking up a lot of space. This section provides a detailed explanation of all UI elements. See Figure 21.

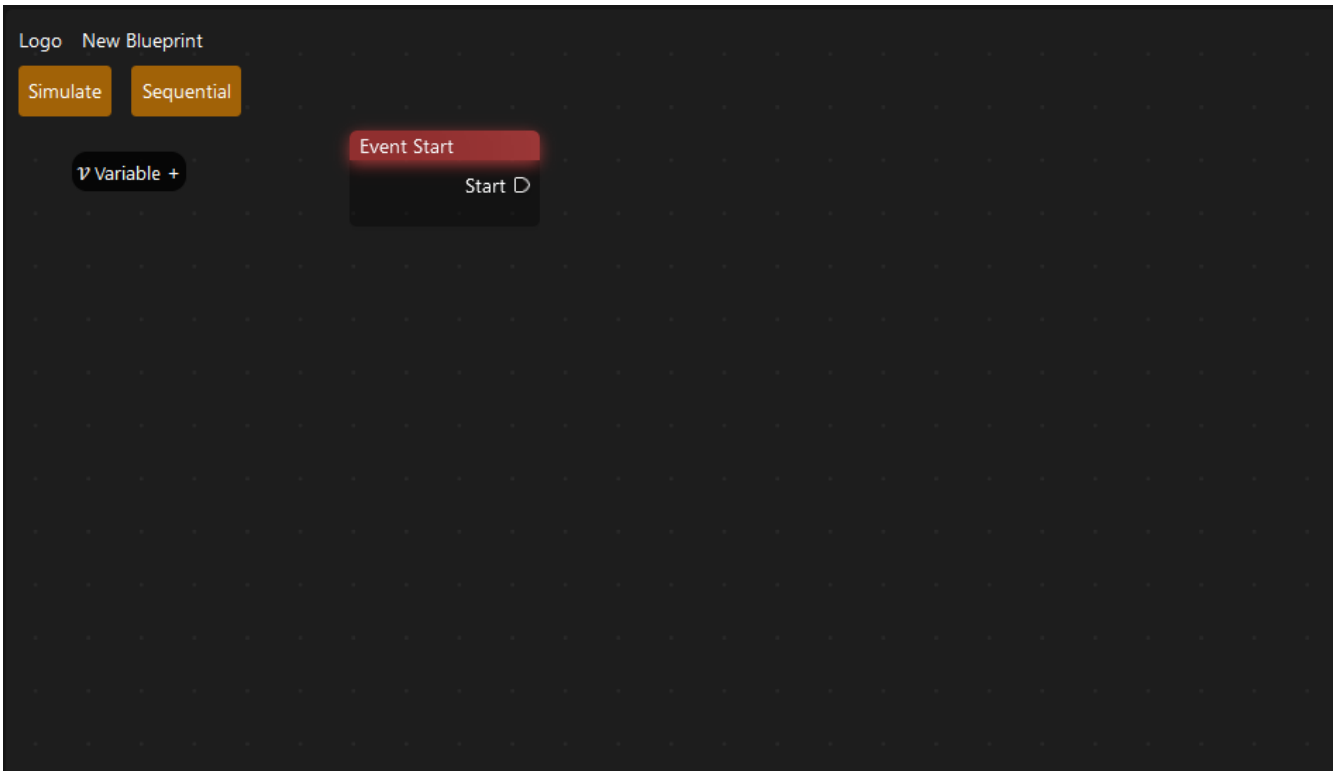

Figure 21: How CVPL looks at launch

4.1.1 Viewport

the Graph is where the user can place, move, and connect the nodes. By right-clicking on the graph, a context menu containing all the nodes the program has to offer will appear. When the user selects one of the nodes, the context menu disappears and the requested node will be placed in the graph where the user's mouse is.

Node The user can drag the node around by holding the left mouse button. The user can drag out connection lines by left-clicking from any of the pins.

4.1.2 Side Panel

The side panel contains multiple UI elements. They are as follows:

Graph Information Element With a default value of "New Script," it displays what script we are currently at. It is there for decorative purposes now.

Simulation Element This UI element contains the Simulate Button and the Simulation Options button. The user can select the scheduler's behavior by selecting the different simulation options. When they are satisfied with the option, they can start the simulation by pressing the simulate button.
Starting the simulation will cause the application to darken, highlight the current node, and display the location of each token. Unless the user selected the Manual Simulation option, the user cannot interact with anything except the Stop Simulation Button. In case they are in the Manual Simulation Option, the user can then interact with the tokens to progress the simulation.

Variable Panel Shown in Figure 22, The Variable Panel allows the user to create new variables. The variables are strongly typed and only mutable when the user declares them to be. To make a variable mutable, the user needs to press the lock icon. By selecting the colored pill button, the user can change the variable type. By default, the variable is set as an integer. Pressing any other area on the variable button displays a pop-up panel, enabling the user to change the default value and edit the name.

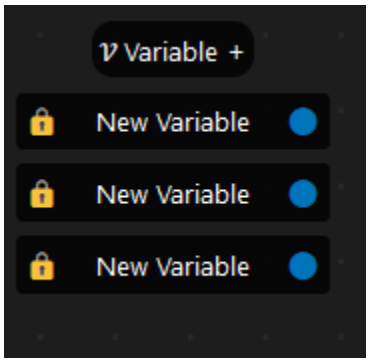

Figure 22: Variable Panel with dummy variables

Console Log Window The console log shows information to the user printed by the print nodes and also any program state information. When starting the program, it welcomes the user and explains the controls. The string "Starting Simulation" appears when the user starts a simulation. If there are print nodes, the prints will be shown after that. Once the simulation ends, it prints "Simulation Complete!".

The user interface in a node-based visual programming language should prioritize a clean and unobtrusive design that allows users to focus on coding. The UI elements described above provide essential functionality, organization, and feedback to enhance the coding experience.

4.2 Components

As visual programming is all about giving clear, visual tools to the user to create programs, it is highly important to keep the tools at hand as visually clear as possible.

4.2.1 Node

By right-clicking anywhere in the grid view, a Node list appears where the user can add a node of their choice. The data regarding each node is stored in the NodeList component and is used by the Node component to create the different nodes when selected. Each node consists of the following data:

1. Name: The name that is displayed on the node.
2. Type: The type of the node, used for decorative and functional purposes.
3. Inputs: Input pins that allow the node to receive data from other nodes.
4. Outputs: Output pins that allow the node to send data to other nodes.
5. Variable: If the node represents a variable, it stores a reference and value to the Variable list.
6. Position: The current position of the node in the viewport.

When a user selects a node from the Node list, the Node component retrieves the selected attributes from the NodeList and applies the appropriate styling to the node in the viewport. It then sends a copy of the node to the NodeList hashmap in the `simulation.ts` file. This hashmap is used to keep track of all the nodes present in the viewport.
Nodes play a crucial role in visual programming as they represent different operations, functions, or variables within the program. They allow users to construct the desired logic and flow of their program by connecting and manipulating these nodes. The attributes associated with each node provide the necessary information for their visual representation and functionality.
Once a Node is created The Node Component can do the following;

1. Moving around the viewport.
2. Creating lines from output pins to input pins or vice versa.

4.2.2 Connection Lines

Connection lines serve as the pathways that tokens use to move from one node to another within the visual programming environment. To ensure safe routing and maintain consistency in the program's logic, certain restrictions are in place for connection lines. These restrictions include:

1. A connection line cannot start and end from Output to Output or Input to Input.

2. Neither can it go from Exec type to Variable type.

3. Connection lines can sometimes convert from one variable type to the other:

   (a) Boolean to String

   (b) Integer to String

By enforcing these restrictions, the connection lines ensure that the program follows a well-defined structure and maintains the integrity of data flow. They contribute to the overall organization and coherence of the visual programming environment, enabling users to create efficient and reliable programs.

4.2.3 Mutex Zone

The Mutex Zone is a red box overlay that enforces tokens to continue traversing through the nodes until they exit the zone. It is represented by a rectangular shape with a red color. When a node is completely within the Mutex Zone, it is considered to be part of the zone.
The mutex zone is a feature in the VPL that can be overlaid on a section of code. When a token enters the mutex zone, it is forced to continue executing each node until the token leaves the mutex zone.
Mutex zones can be overlaid on top of eachother. each n
Upon creation of a Mutex Zone, a number appears on the top right-hand corner of the zone, indicating its depth. This number helps indicate the nesting level when there are multiple Mutex Zones within each other. It allows users to identify how deep they are in the Mutex Zone hierarchy.
The purpose of the Mutex Zone is to control the flow of tokens within a specific region of the program. Tokens entering the Mutex Zone will continue to traverse through the nodes until they exit the zone. This ensures that certain sections of the program are executed exclusively and in a specific order, preventing concurrency issues or conflicts.
The Mutex Zone is a useful tool for managing the synchronization and coordination of nodes within a visual programming environment. It allows for precise control over the execution flow, ensuring the desired logic and sequencing of operations are achieved.

4.3 Simulation

When developing a visual programming language, there are different approaches to executing the code represented by the visual components. One approach is to have a close 1:1 code translation, where the visual components are directly converted to actual code. For example, Unreal Engine's Blueprint Language translates nodes to C++ code during the build process. This approach maintains performance similar to writing code directly in a programming language but requires implementing a translator to accurately convert each visual element to its text-form counterpart, which can be time-consuming.
In this Proof of Concept, we have chosen a different approach: designing our own language. The goal of this research is to showcase the benefits of a visual programming language with concurrency. To maintain focus on this objective, we have created a simple Turing-complete language. Although it can be expanded to include more functions and complexity, we envision using Rust as the target language for the visual programming language translation in the future.

In this section, we will explore how the simulation works and how concurrency is enabled within the visual programming language.

4.3.1 Token

A token can be visualized as a coin that moves from node to node, following the appropriate connection lines until it can no longer proceed. Tokens are data structures that store the following information:

4.3.2 Scheduler

The scheduler is responsible for controlling the execution flow within the visual programming language. Its operation can be summarized as follows:

1. Focus on the node where the token is currently located.
2. Perform the node's function if it has not been executed yet.
3. Traverse to the next node if possible; otherwise, the execution is completed.

Simulation Options The simulation options allow the user to change the concurrency model used in the program execution.

Round Robin The round-robin scheduler sequentially lets a token move and execute the next node.

Lottery Based The lottery-based scheduler gives each token a ticket and then randomly selects a ticket. The token that holds the selected ticket is allowed to move and execute the next node. The remaining tokens keep their tickets and the scheduler repeats itself until completion.

Manual The manual scheduler allows the user to decide which token will move and execute the following node. This allows the user to visualize any possible order of execution the concurrent program can have.

4.3.3 Memory

The VPL Stores the Nodes and Variables in simple Maps and Tokens to an array.

4.3.4 Actions

Actions are the Functions Nodes refer to when the Token commands the Simulation to perform an action. An action can only be done once by a node. The following is a comprehensive table of all the actions for each node.

Action Name	Used by Node	Description	Parameters	Result
Branch	Branch	If condition is true, follow the true path otherwise follow false	token, node	token.outputIndex is set to either 0 or 1
For	For	Loops from start until end	NA	NA
Fork	Fork	Pushes a new token to the TokenList	token.key	new token starts at the Fork node
Join	Join	Discards Children tokens and continues with the Parent threads	NA	NA
Print	Print	Displays string in the console log	token, node	Prints the string parameter in the browser's console log
Add	Add	Sets the output value to the sum of the values of the input connection lines/input boxes	token, node	output value = sum of input values
Set	Set	Sets the variable to given input variable	token, node	
Get	Get	get value from variable		

5 Discussion

This section discusses the benefits and limitations of the proof of concept concurrent VPL.

5.1 Summary and Interpretations

The VPL's approach to concurrency through simulating parallelism provides a valuable learning tool for new developers to understand how concurrency works. While the VPL may not be directly applicable to real-world applications, it can be used as a complementary tool alongside textual programming to explore and experiment with concurrency concepts.

5.1.1 Defining the VPL

The proof of concept is a node-based VPL inspired by both Unreal Engines Blueprint VPL and Blender Geometry Nodes VPL. By linking up nodes, a program can be made and simulated through a multitude of schedulers. The VPL is aimed to be an educational tool first, helping users to understand concurrency concepts in an intuitive and efficient manner.

5.1.2 The basics

While textual-based programming languages such as C are still easy to understand when creating small simple programs, they still have a barrier of entry as users are forced to follow a specific syntax. With a VPL, the entry of developing simple programs such as "Hello, World" is minimal, and enables more users to explore further.

5.1.3 Variables and Loops

Variables and loop enables programming languages to be Turing-complete. This means that any program can be made using the program given enough time and memory. VPLs have the benefit of keeping a good overview of the branches caused by if statements and loops. This might hamper the user's optimization skills when going back to programming textually as this VPL does not help the user with gaining a habit of maintaining code clarity.

5.1.4 Concurrent Programming

Concurrent Programming enables developers to use all the threads the CPU has. It allows for efficient programs to be made, as long as the communication between the threads is well set. This is a tough task to achieve as the cognitive load increases the more threads there need to be kept in mind. Through a VPL however, the branchings of threads are clearly shown and by zoning off certain parts of the code through the use of mutex zones, a developer can develop a concurrent program more effectively.

5.2 Expected Benefits

The proof of concept of a concurrent VPL is expected to enable users to quickly understand concurrent programming and how different schedulers affect their programs. Previous research has shown that because of the nature of VPLs allowing for instant feedback and a closer relation between the concepts thought and the coding workflow, users will be able to rapidly understand concepts such as mutex locks and forking.

5.3 Limitations

In its current state, the VPL may not meet the needs of every developer. It is primarily designed as an educational tool for introducing concurrency concepts. Further, the VPL does not offer profilers or advanced debugging tools. However, it does empower developers to gain a deeper understanding of concurrency and visualize its execution. Users can gain insights into how different scheduling algorithms may impact their program's performance and fairness by manually manipulating the flow of threads and observing the resulting program behavior. This knowledge can inform their decision-making when implementing concurrency in real-world applications.

In conclusion, the VPL serves as a valuable tool for learning and experimenting with concurrent programming concepts. Providing visual cues and options for manual control helps developers reduce cognitive load and better understand how concurrency can be effectively utilized. While not a standalone solution, the VPL can complement textual programming and serve as a stepping stone toward mastering concurrent programming concepts.

6 Conclusion and Future Work

In conclusion, the CVPL serves as a valuable educational tool for learning and experimenting wi th concurrency concepts. While it may not be directly applicable to real-world applications, it offers a valuable learning experience for new developers to understand how concurrency works. The VPL's node-based approach provides an intuitive and efficient way to explore and experiment with concurrency concepts.

The VPL has several benefits, such as simplifying the understanding of concurrency for users. It lowers the barrier of entry for developing simple concurrent programs and enables them to see the effects of the different provided schedulers. Concurrency programming is facilitated through the VPL by the fork node, mutex zones, and scheduling options.

While the VPL has its limitations, primarily being an educational tool and lacking profilers and advanced debugging tools, it empowers developers to gain a deeper understanding of concurrency and visualize its execution. By manually manipulating the flow of threads and observing the resulting program behavior, users can gain insights into the impact of different scheduling algorithms on performance and fairness.

Overall, the concurrent VPL complements textual programming and serves as a stepping stone toward mastering concurrent programming concepts. It provides visual cues, reduces cognitive load, and helps developers better understand and utilize concurrency in their programs.

6.1 Future work

There are plans to improve the CVPL. Developing the functionality of the for loop node and join node are ones aimed to be done first. After that, the ability to create functions with nodes will be implemented. This would allow users to develop more complex code. Continuous improvement in user comfort will also be considered once the previous points have been developed.

References

[AB18] Syed Ahmed and Mehdi Bagherzadeh. What do concurrency developers ask about?: a large-scale study using stack overflow. 2018.

[AT05] T. Ajiro and K. Tsuchida. A bit-level concurrent visual programming language (a-bits) and a base computation model (apc) for its development. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 269–271, 2005.

[Ble] Shader editor. https://docs.blender.org/manual/en/latest/editors/shader_editor.html.

[ES6] Promise. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

[ImG] Dear imgui. https://github.com/ocornut/imgui.

[LK14] Sze Lye and Joyce Koh. Review on teaching and learning of computational thinking through programming: What is next for K-12? *Computers in Human Behavior*, 41:51–61, 2014.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1980.

[Mil99] R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, Cambridge, UK, 1999.

[Nel55] Raymond J. Nelson. Review: D. A. Huffman, the synthesis of sequential switching circuits. *Journal of Symbolic Logic*, 20(1):69–70, 1955.

[Rei13] Wolfgang Reisig. *Understanding Petri Nets*. Springer, Berlin, 2013.

[RMMH+09] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for all. *Communication of the ACM*, 52(11):60–67, 2009.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall PTR, USA, 1997.

[SC21] Ana Luisa Veroneze Solórzano and Andrea Schwertner Charão. Blocklypar: From sequential to parallel with block-based visual programming. In *2021 IEEE Frontiers in Education Conference (FIE)*, pages 1–8, 2021.

[Sch97] R.R. Schaller. Moore's law: Past, present and future. *IEEE Spectrum*, 34(6):52–59, 1997.

[Sve] Sveltekit. https://kit.svelte.dev/.

[Tau] Tauri 1.4. https://tauri.app/.

[TCT97] S.J. Turner, Wentong Cai, and Hung-Khoon Tan. Parallel programming with VPE: A case study of an integrated visual programming environment. In *Proceedings High Performance Computing on the Information Superhighway, HPC Asia '97*, pages 319–324, 1997.

[Tsa18] Chun-Yen Tsai. Improving students' understanding of basic programming concepts through visual programming language: The role of self-efficacy. *Computers in Human Behavior*, Volume 95, 2018.

[Unr] Introduction to blueprints. https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/GettingStarted/.

[Vul] Vulkan. https://www.vulkan.org/.