Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Visual Pi-Calculus

Design and Implementation of a Block-based Language for Modelling Pi-Calculus Systems

Dominik Guzowski

Supervisor: Dr. Vasileios Koutavas

A dissertation
submitted to Trinity College Dublin, the University of Dublin in partial fulfilment of the requirements for the degree of
Master in Computer Science

April 2024

Declaration

I hereby declare that the following dissertation, except where otherwise stated, is entirely my own work, and is not work generated through the use of any GenAI tools; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

Dominik Guzowski
29 April, 2024

Acknowledgements

I would like to express my sincere thanks and gratitude to my supervisor, Dr. Vasileios Koutavas, for his support, discussions, explanations and advice throughout the course of this dissertation. My meetings with Dr. Vasileios Koutavas have helped shape the design of the language presented in this paper and inspired many design choices taken throughout the language design process. Without his guidance and mentorship, the completion of this dissertation would not have been possible.

Abstract

Complex concurrent systems are often modelled using formal mathematical frameworks called process calculi, such as pi-calculus, to facilitate a deeper understanding of their complexities. These frameworks enable the analysis and reasoning of concurrent systems. Representations of the system interactions can be depicted through directed graphs, such as labelled transition systems, which aid in visualising and exploring the possible states and transitions within the system. PIFRA, a tool utilising a modified syntax of pi-calculus, generates intricate labelled transition system graphs based on provided models through said syntax.

This dissertation proposes and implements a visual language utilising blocks as an alternative means of constructing pi-calculus models for PIFRA. The language aims to be user-friendly, catering to novices, while retaining the ability to represent any pi-calculus construct. Additionally, it assists in crafting accurate models by ensuring well-defined name scoping and intentional name usage, thereby reducing the likelihood of errors such as typos.

Contents

Nomenclature

AST *Abstract Syntax Tree*
Hierarchical data structure used to represent the structure of source code of a programming language according to its grammar.

UI *User Interface*
The means by which a user interacts with a computer program using a graphical interface.

FRA *Fresh-Register Automata*
A formal model in automata theory used to describe the behaviour of register automata with the concept of freshness – ensuring values are unique.

PIFRA *Pi-Calculus Fresh-Register Automata*
A tool used to generate labelled transition system graphs of pi-calculus systems.

LTS *Labelled Transition System*
Description of a system and its behaviour through a labelled directed graph where states are represented by nodes and transition by edges.

CSS *Cascading Style Sheets*
Computer language used to define styles for laying out web pages.

IDE *Integrated Development Environment*
A software application for software development typically including a code editor and other tools integrated into a single interface.

VisPi *Visual Pi-Calculus*
Block-based language for pi-calculus implemented in this paper, and the name of the web application which hosts it.

1 Introduction

Concurrent and parallel systems are a fundamental paradigm in computer science that is essential to the development and operation of modern systems and infrastructures. These involve the simultaneous execution of multiple tasks or processes, which may communicate with each other to form complex systems whose behaviour may be non-deterministic and hard to fully predict. Despite their prevalence, concurrent and parallel systems are often difficult to fully understand, primarily due to their inherent complexity and the non-intuitive nature of concurrent behaviour. In addressing this challenge, process calculi, notably the pi-calculus [1], are invaluable tools facilitating the modelling and analysis of concurrent and parallel systems. By providing formal frameworks for reasoning about concurrent processes and their interactions, process calculi offer insights into the behaviour of complex systems, thereby enhancing our comprehension and enabling the development of robust and scalable computational solutions.

1.1 Motivation

The pi-calculus, while offering promise in enhancing our ability to reason about concurrent systems, presents a formidable learning curve due to its inherent complexity. Consequently, this complexity often acts as a barrier, deterring individuals from delving deeper into the field.

In addressing this challenge, the development of tools such as PIFRA [2] has proven invaluable. PIFRA enables the generation of labelled transition state graphs from pi-calculus systems, providing a visual representation of the states and transitions within the modelled system. However, while PIFRA facilitates visualisation, it remains reliant on the intricate syntax of the pi-calculus, thereby limiting its accessibility to novices in the field.

Recognizing the need for a more approachable entry point into concurrent computing, this project aims to bridge this gap by introducing a novice-friendly block-based language. This language will serve as an intermediary, allowing individuals to model systems without the prerequisite understanding of the intricate pi-calculus syntax. By transpiling down to PIFRA, this solution seeks to empower novices to engage with parallel and concurrent computing concepts with greater ease and confidence.

Through the development of a block-based language, the goal is to provide an easier access to concurrent computing modelling and analysis, creating a more approachable introduction to process calculi.

1.2 Objectives

Goal of this dissertation project is to design and provide a working implementation of a visual language whose output will be a valid PIFRA program. The language is to be implemented in a user-friendly web application that is easily accessible.

The overall objectives are:

1. Design and implement a block-based language for pi-calculus.
2. Transpile from blocks to valid PIFRA code.
3. Implement a user friendly web application IDE.
4. Add error checking and meaningful error messages.
5. Implement a name-scope safety system.
6. Test the language with existing examples using PIFRA.

In addition to the main objectives, some other stretch goals are:

1. Implement syntax highlighting for the output code to match blocks.
2. Add serialization to save and load programs.

2 Background

This section will cover the visual paradigms used previously for encoding parallel and concurrent programs or distributed systems, a brief overview of the Blockly [3] library and covering the basic concepts of pi-calculus [1] as well as PIFRA [2].

2.1 Related Work

Modelling concurrent systems is a challenging task due to the necessity to represent every behaviour and interaction of many communicating processes. Modelling of such systems through graphical interfaces is equally challenging, especially if we aim to use them to ease our understanding of the complex and dynamic behvaiours of the modelled systems.

Multiple visual approaches have been used for the task, including the use of custom visual languages, visual programming idioms such as blocks or nodes and links or connected-icon-based languages or Unified Modeling Language (UML) diagrams.

The visual representations can be categorised into two main types: variations of *node graph* architectures and *block-based* languages. Node graphs use nodes connected by edges or links to represent connections between processes, and often provide an explicit visual representation of the system topology where each process is connected to other processes it communicates with. Block-based languages use predefined blocks which are dragged around the screen, such as Scratch [4]. Block-based languages in the context of parallel system modelling generally focus on the behaviour of a single process and do not visually connect communicating processes together. They also use event-based blocks to describe communications between processes [5] which are blocks following the pattern `when <event> do`.

2.1.1 Uppaal

Uppaal [6], developed collaboratively by Aalborg University in Denmark and Uppsala University in Sweden, is a tool for modelling, simulating and

verifying real-time distributed systems. It is used for modelling non-deterministic processes with finite control structures and real-valued clocks, which communicate through the use of channels or shared variables.
The tool consists of a description language, a simulator and model-checker. The description language allows modelling of the system behaviour through the use of extended automata with clock and data variables. The simulator is used for validation of the system's behaviour and the model-checker verifies the properties of the system by exploring the system's state-space.
The visual representation used in Uppaal is a timed automata graph, with nodes representing the states with the transitions between states being encoded through directed edges.

2.1.2 NetsBlox and Snap!

NetsBlox [5] is a visual programming language designed to facilitate the creation of distributed applications. The visual idiom employed is blocks, an extension of the open source Snap! [7] which itself is an extended reimplementation of the Scratch [4] visual programming language.
NetsBlox abstracts away much of the complexity involved in distributed systems programming, such as networking and communications protocols. However, it serves as a relevant example of how block-based languages can be effective at representing complex non-linear programming patterns and concurrent, distributed system communications.

2.1.3 Other Parallel Visual Languages

Parallel visual languages have been used for many domains involving distributed applications and systems. Kaira [8] is a visual modelling tool which uses a variant of Coloured Petri Nets (CPNs) to model, simulate and generate parallel applications. CPNs visually resemble activity diagrams in UML and for the purposes of visual categorisation, are a variation of a *node graph*. Some other examples of visual, graph-oriented languages include VisualGop [9] and VPPE [10]. VisualGop is a programming environment aiming to provide high-level abstractions for configuring and programming parallel processes through the use of graphs representing communication and synchronisation between programs. VPPE is a cloud environment incorporating icons to encode operations such as input-output, workflow, communication and processing.

2.2 Creating Block Languages with Blockly

Blockly [3] is an open source library developed by Google, which allows for incorporating a block-based editor in a web application. Blockly does not provide a runtime directly and only provides code generation to a target language, leaving the developer to decide whether and how to run the code. By default, Blockly provides block vocabularies and generators for a variety of commonly used programming languages, such as JavaScript and Python, which can be used directly without any additional setup within a web application. Blockly also provides a way for creating custom languages, the blocks for those languages, and a code generation API for defining how blocks translate to code.

E. Pasternak et al. [11] provide many important tips for designing languages with Blockly. Some of the main tips include the audience, scope and language used. The audience is the age and experience of the intended user base. These could include young children learning to code, college students using the language for learning or even IT professionals using block-based languages for their day-to-day work. The scope refers to limiting the amount and complexity of the blocks which are available to the user. Too many blocks may impede usability and make it harder for the user to find what they need. Finally, the language used in the blocks. This could be natural language, making the text in the blocks read as sentences, or computer language is closer to the true output language. These considerations aim to remove common barriers to entry such as the discoverability of language features of the target language [12] [13].

2.3 The Pi-Calculus

The pi-calculus is a process calculus developed by Milner and is an extension to the Calculus of Communicating Systems [14], which has also been developed by Milner, with the aim to address the shortcoming in relation to modelling dynamic systems and their communications.

A process calculus or algebra, is a formal language used for modelling, checking and verifying the models of concurrent systems through a set of rules, with the aim of reasoning about the behaviour and relationships between concurrent processes. The pi-calculus has many variations, however the examples in this paper follow the formalisations based on Milner's book 'Communicating and Mobile Systems: The Pi-Calculus' [1].

2.3.1 Syntax and Notation

At the core of pi-calculus are *names*, which are generally lowercase identifiers $a, uv, xyz \ldots \in N$. Names serve multiple purposes, such as indicating data or communication channels. We will often use *names* or *channels* interchangeably depending on context.
Names can often be thought of as variables in programming languages, which are just identifiers and do not necessarily indicate what data is stored within them. In addition to names, pi-calculus uses *processes*, usually indicated through uppercase identifiers, with their own defined behaviour.

For the purposes of this paper, only the basics of the pi-calculus will be explored to the extent they are relevant to the design and implementation sections of the paper. Additional theory which is not immediately relevant shall be mentioned briefly or omitted.

$P, Q ::=$	*process definition*
$a(b).P$	*input*
$\bar{a}\langle b\rangle.P$	*output*
$\nu a\ P$	*restriction*
$P + Q$	*summation*
$P \mid Q$	*composition*
$P!$	*replication*
0	*inaction*

Table 2.1: Pi-Calculus syntax constructs

2.3.1.1 Replication

$P!$ is the indefinite creation of copies of the same process P. This provides a way to encode persistence of the process P.

2.3.1.2 Inaction

Inaction or termination of a process is denoted by 0, or the nil process. It marks the end of the process.

2.3.1.3 Input – Receiving names

$a(x).0$ is the notation used to encode the name x being received on channel a, followed by the inaction marker 0. Receiving introduces a name to the scope in which it is received, and binds it to the name x.

Example A: Input

Consider the pi-calculus expression $a(b).b(x).0$. Now suppose a channel c was sent on channel a, then received on channel a in our expression, resulting in the substitution for c in all occurrences of b such that the remaining part of the expression becomes $c(x).0$.

2.3.1.4 Output – Sending names

$\bar{a}\langle x\rangle.0$ is the notation used for encoding a name x being sent across channel a followed by the inaction marker 0. A $locally$ bound name in a process can be sent across known channels and be received by a different process waiting to receive a name on some known channel, establishing a way for encoding communication between processes.

2.3.1.5 Composition – Parallel execution

$P \mid Q$ denotes a parallel composition of two processes P and Q. Both processes will run simultaneously.

Example B: Composition

Consider the expression $\bar{a}\langle x\rangle.x(b).0 \mid \bar{a}\langle y\rangle.y(c).0 \mid a(z).\bar{z}\langle w\rangle.0$. Here we have described three processes (call them $X \mid Y \mid Z$) running in parallel with a race condition where two processes are sending on the same channel a, while a third process is waiting to receive on the channel a. Depending on whether x or y is received on the channel a as the name z, the process whose name was not received will stall indefinitely until its name is received. This expression can collapse into two different outcomes:

Outcome 1: $w(b).0 \mid \bar{a}\langle y\rangle.y(c).0$
Process X won the race condition and received the w on channel x, and is ready to receive b on channel w.
Process Y lost the race condition and remains waiting for its message to be received.
Process Z has completed its execution by receiving x on channel a and then sending w across x.
Outcome 2: $\bar{a}\langle x\rangle.x(b).0 \mid w(c).0$
Process X lost the race condition and remains waiting for its message to be received.
Process Y won the race condition and received the w on channel y, and is ready to receive c on channel w.
Process Z has completed its execution by receiving y on channel a and then sending w across y.

2.3.1.6 Summation – Non-deterministic choice

$P + Q + R$ represents a choice that is non-deterministic in nature, meaning that out of the processes P, Q and R, only one will ever execute and never more than one.

Example C: Summation

Consider the expression $\bar{a}\langle x\rangle.0 + a(y).0$. Here we have two processes, both of which are attempting to act on channel a. If the summation was a parallel composition, we would have both processes communicate with each other by sending x across a and then receiving it as y, since only one process in a summation can execute, these processes will never communicate as only one can execute, but never both.

2.3.1.7 Restriction – Introducing a locally bound name

νx or $new\ x$ introduces a new *bound name* to some current process scope P which is denoted by x. In some other process Q, x does not exist and references to x is Q refer to some *free name* x.

A *free name* is a globally known name which different processes can reference and interact with, while a *bound name* is privately scoped to the process to which it is restricted to. Understanding the difference can be done through an analogy with the C programming language.

Example D: Free names vs. Bound names

```
int x = 0;

void Foo() { int x = 0; x = 2; }
void Bar() { x = 5; }
```

Listing 2.3-1: Free name vs. Bound name analogy using C.

We defined a global variable `x` and two functions, `Foo` and `Bar`. In this example, the function `Foo` introduces a local variable `x` with the value `0` and proceeds to assign it immediately with the value `2`. Neither action done by `Foo` has interacted with the global `x`.
The function `Bar` simply assigns the variable `x` with a value of `5`, however it never introduced a local variable, therefore `bar` has interacted with the global variable `x`. In this example, `Foo` acts on a bound name x, which shadows the free name x whereas `Bar` acts on a free name x.

Example E: Restriction

Consider the following expression:

$$\nu x\ \bar{a}\langle x\rangle.\,x(y).\,0 \mid \bar{a}\langle x\rangle.\,x(z).\,0 \mid a(b).\,\nu c\ \bar{b}\langle c\rangle.\,0 \mid x(w).\,0$$

Here we have four processes running in parallel, let's call them in order $A \mid B \mid C \mid D$.
Process A restricts a new *bound name* x in its scope, then proceeds to send it across the free channel a and finally waits to receive some data on the private channel x as the name y.
Process B sends a *free name* x on the free channel a, then proceeds to wait to receive data on the free channel x as the name z.
Process C waits to receive a message as the name b on channel a. Once received, proceeds to restrict a new name c and send it across the received channel b.
Process D simply waits to receive a message on the free channel x.
Here we have an evident race condition between processes A and B for which can send their message across a first. If A wins, processes B and D stall indefinitely in a deadlock, as process C is able to send the message across the private channel created by process A.

However, if process B wins the race condition, process A immediately stalls, but another non-deterministic scenario arises between processes B and D. Since B sent a free x across a and then proceeded to listen on that same x, it has no guarantee that if it managed to communicate with C, that it will receive a response on the channel it sent. This is because we have process D which is constantly waiting to receive a message on the free channel x and can intercept the message sent by C. Therefore, either process B or D will stall, depending on which manages to receive the message from C. This example can be viewed as a secure connection communication (A communicating with C) and a non-secure connection communication (B communicating with C) since D can intercept the response from C due to it having knowledge of the free channel x.

2.3.2 The Extended Pi-Calculus

The extended pi-calculus [15] is an extension to the pi-calculus syntax to support its representation through fresh-register automata. For the purposes of this paper, we shall only discuss the relevant additions made to the supported constructions of the pi-calculus.
Tzevelekos introduced the equality construct, and in addition to the equality construct, the inequality construct was introduced [2] along with minor changes to the base pi-calculus syntax.

$P, Q ::=$	*process definition*
$a(b).P$	*input*
$\bar{a}\langle b\rangle.P$	*output*
$[a = b]\ P$	*equality*
$[a \neq b]\ P$	*inequality*
$\nu a.P$	*restriction*
$P + Q$	*summation*
$P \mid Q$	*composition*
$P(\vec{a})$	*process call*
0	*inaction*

Table 2.2: Extended Pi-Calculus constructs, with the inclusion of the dot post a restriction [2].

Note the addition of the dot in the restriction construct. The process call $P(\vec{a})$ replaces replication $P!$ as it provides the same ability for a process to persist.

2.3.2.1 Process Call

P or $P(\vec{a})$ invokes a defined process, which can be thought of as a function call in standard programming languages. $P(\vec{a})$ is the invocation of a process with a vector of names as arguments to the process.

Example F: Process Call

Consider the following:

$$P(x) ::= \bar{a}\langle x\rangle.\, \nu y.\, P(y)$$
$$\nu z.\, P(z)$$

Here we have defined a process P which is parameterised with the name x. P is recursive on itself, and with each invocation it broadcasts the name across channel a. It then generates a fresh name and calls itself with the fresh name, continuously broadcasting unique names across channel a. The first name broadcasted will be the bound name z followed by an infinite stream of unique names thereafter.

2.3.2.2 Equality and Inequality

$[a = b]\, P$ and $[a \neq b]\, P$ are the constructs for equality and inequality respectively. These are equivalent to *if conditions* in standard programming languages. The conditions are evaluated after the name substitution step and the process P will only execute if the condition evaluated as true, that is the names after the substitution are either equal or not equal.

Example G: Equality and Inequality

Consider the following system:

$$P(b) ::= \nu x.\, \bar{a}\langle x\rangle.\, b(y).\, [x = y]\, P(b)$$
$$Q(b) ::= a(z).\, \bar{b}\langle z\rangle.\, Q(b)$$
$$\nu c.\ (P(c) \mid Q(c))$$

In the system we have two recursive processes, P and Q, both communicating on a public channel a and and a private shared channel c. Process P continuously creates new names x, broadcasts them on channel a and the proceeds to receive a message on channel b (channel c after substitution). It then verifies that the received name y is the same as the broadcasted x, if so, P invokes itself again, otherwise it terminates.

Process Q simply continuously receives names on a and forwards them on b. In this example P and Q are in constant communication with each other, however in the instance that some other process sent a message across channel a, process Q would forward an unknown name to P resulting in the termination of P, followed by the stalling of Q thereafter due to channel b no longer having a process waiting to receive messages.

2.4 PIFRA

PIFRA [2] is a command-line tool which uses fresh-register automata [15] to generate labelled transition systems based on pi-calculus models, for model checking and verification whether certain states in the system are reachable. The language used to write the models is an ASCII representation of the extended pi-calculus. Given that the syntax is very close to pi-calculus syntax, which is quite abstract in nature, it can be relatively difficult to start using without a proper introduction into process calculi and the pi-calculus syntax. PIFRA as a tool does not have a dedicated environment, syntax highlighting or detailed error reporting.

Extended Pi-Calculus		**PIFRA ASCII**
$P ::=$	*process definition*	`P =`
$Q(a,b) ::=$	*parameterised process def.*	`Q(a, b) =`
$a(b).P$	*input*	`a(b). P`
$\bar{a}\langle b\rangle.P$	*output*	`a'<b>. P`
$[a = b]\,P$	*equality*	`[a=b] P`
$[a \neq b]\,P$	*inequality*	`[a!=b] P`
$\nu a.P$	*restriction*	`$a. P`
$P + Q$	*summation*	`P + Q`
$P \mid Q$	*composition*	`P | Q`
P	*process call*	`P`
$Q(a,b,c)$	*parameterised process call*	`Q(a, b, c)`
0	*inaction*	`0`

Table 2.3: The PIFRA ASCII equivalents of the extended pi-calculus constructs.

3 Design

The design section will cover the decisions and rationale behind certain choices made in the actual design process of the VisPi visual language and the web app IDE, and serves as an introduction to the implementation section that follows.

3.1 Choice of Visual Representation

VisPi first and foremost is intended to be an introductory visual language to pi-calculus and modelling concurrent systems, and a tool used for general concurrent system modelling second. That being said, the language must be simple and easy to learn and use, and for that reason the visual paradigm employed in VisPi is block-based due to its simplicity, approachability and discoverability of blocks for novices to the language [12], minimising the barriers of entry into pi-calculus.

3.2 Language Design Principles

When designing blocks with Blockly [3], the principles outlined by Pasternak et al. [11] were considered to ensure the design of the language is appropriate for the target audience, is appropriately scoped in terms of number of blocks, uses appropriate and familiar language and most importantly is consistent.

3.2.1 Target Audience

The primary target audience is computer science and engineering-oriented students in third level education and novices to concurrent system modelling, analysis and process calculi in general. The secondary audience is professionals working with concurrent system modelling for whom VisPi would serve as an alternative environment for writing models. Given the audience is individuals who are expected to have basic computer science knowledge and some notion of parallelism and concurrent programming, the language design does not need to cater to younger audiences and can be more abstract and high-level as a result.

3.2.2 Set of Blocks

Given the main aim of the language is to be an introduction to pi-calculus, the scope in terms of number of blocks should remain relatively low, to ensure that it does not become cumbersome to find the required blocks the user may need as to not impede on the workflow.
For that reason, the blocks shall be limited to the bare minimum required to capture the entirety of pi-calculus, with the exception of block variants which would allow users to write programs more concisely and reduce duplication.

3.2.3 Language in Blocks Labels

There are two main approaches to the language that should be included in each block, those being natural language or computer language [11]. Natural language refers to using sentence-like labels on blocks, which read like English and express more complex notions than icons or symbols or the literal language syntax. Computer language on the other hand is just that, the literal use of the target transpiled language syntax on the blocks themselves. Given the concise nature of pi-calculus syntax, the use of computer language is not feasible but also does not address the issue of learning pi-calculus' abstract syntax.

3.2.4 Consistency

To ensure that using the language is intuitive, we shall use the same colour and language for blocks which serve the same purpose, or are closely related, as well as ensuring the shape of the blocks is also consistent depending on the block's usage.

3.2.5 Good Defaults

Blocks which require the use of names, or often or always are paired together, should be by default populated appropriately. This serves two purposes, improving the usability of the language and the agility at which it can be used, but also provides examples of the language patterns from the get-go and teaches the user how blocks should be connected.

3.2.6 Additional Design Considerations

Since one of the goals for VisPi is introduction to pi-calculus, it is important that we try to eliminate as many possible points of error as we can and provide implicit hints to the user about the syntax and semantic meaning of the programs they model to avoid ambiguous scenarios.

3.2.6.1 Free Names

One of the strengths of pi-calculus is the unrestricted use of names, without the need to pre-define them [1]. While providing flexibility and expressiveness while modelling concurrent systems and their communications, this aspect of pi-calculus may pose two problems in a visual environment:

1. *What is 'x'?*

   To someone who is unfamiliar with the syntax of pi-calculus, a random identifier thrown into a model may be confusing initially. Additionally, in a visual interface where we are required to somehow provide textual input, avoiding unnecessary typing may reduce the possibility of typo errors.
2. *What was the name I used in that process?*

   In a visual setting it is expected that we will not fit as much information onto the screen as we would otherwise with pure text. This introduces the need to move around in the block editor to potentially find instances where a certain free name was introduced for a common communication channel between processes, and this problem grows the larger the system we are modelling.

To address this issue, we propose the language to include a block which does not produce code directly, but is used to define free names by the user. In addition to that, in order to reduce the chance of user making a hard-to-spot typo, using names in programs shall be done through a dynamic dropdown. This dropdown would include the available names which have been defined by the user.

3.2.6.2 Name Scoping

Parentheses are, in some cases, optional in pi-calculus, and sometimes are used intentionally to define explicit scopes. This can lead to hard-to-spot errors especially in the textual format such as the following:

```
$x. a'<x>. 0 | b'<x>. 0
$x. (a'<x>. 0 | b'<x>. 0)
```

Listing 3.2-1: Using parentheses to change the scoping of a name.

The difference between the two systems is the scope of x. The first system defines x only in the left-hand side process, whereas the the right-hand side uses a free name x, which is semantically different from the latter system where x is a bound name in both parallel processes. In fact, to better illustrate the difference, the first system can be written as

```
($x. a'<x>. 0) | b'<x>. 0
```

Listing 3.2-2: Alternative representation of the first system in **Listing 3.2-1**.

To prevent the possibility of such errors occurring unintentionally, we shall enforce that all blocks which introduce bound names into the system to *wrap around* the subsequent blocks. This would visually encode the fact that the newly introduced name is bound within the confines of the block which introduced it.

Additionally, we shall enforce that the aforementioned name dropdown to only contain the names which are in-scope at any given point in the program, preventing the possibility of selecting an incorrect name which is introduced elsewhere in the system as a bound name.

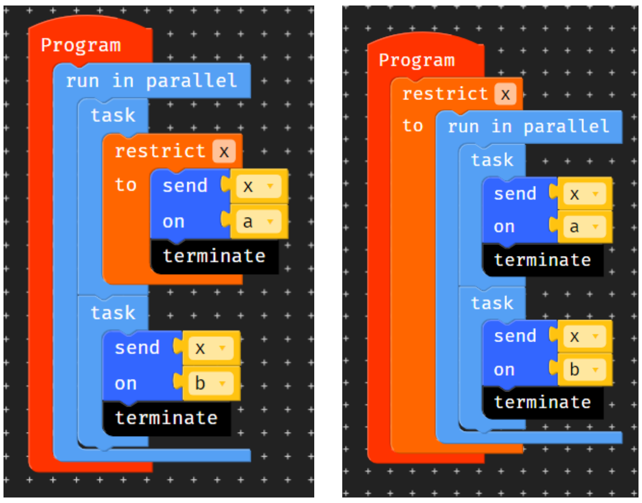


Figure 3.1: VisPi Blocks representing the processes in **Listing 3.2-1**.

3.2.6.3 Use of Colour

Aside from ensuring blocks serving a similar purpose or being related having matching colour, the same colours shall be used in the PIFRA code output. This will serve the purpose of implicitly teaching the syntax through visual cues, for example all *send* or *output* actions being coloured blue, both in blocks and in the output code.

3.3 Encoding Non-Sequential Execution

Representing parallelism in blocks is not an easy task unlike with nodes and links where we can use many links from a single node to encode parallelism. There are three main ways of assembling blocks with Blockly [3], those being *next statement*, *value input* and *statement input.*

Next statement refers to blocks that snap onto the bottom of another block vertically in a sequence.

Value input refers to blocks which attach horizontally, in a puzzle-piece-like manner. These can also be inserted inside the actual structure of a block instead of always being attached to the side of the block. This can be used as a way to insert variables or expressions into other blocks, such as *if statement* conditions.

Statement input refers to a block which accepts an entire sequence of blocks inside of itself, and is a form of nesting. Just like *value input* can be used as the *if statement* condition expression, *statement input* is used for the body of the *if statement.*

We shall discuss how we could use each of the three assembling methods to represent non-sequential execution, as well as the advantages and disadvantages each method brings.

3.3.1 Blockly List Value Input

Blockly [3] provides a way for constructing dynamic list blocks, which allow other blocks to attach as value inputs, that is, blocks which attach to the side, rather than as the next block vertically, or a nested block.

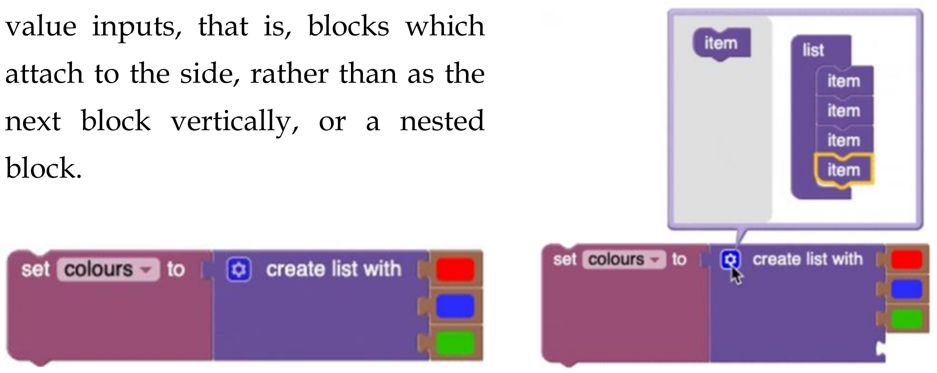


Figure 3.2: Blockly list block and the configuring of the number of accepted items in the block.

Additional slots can be added to the list block by adding additional *item* blocks to the *list* blocks in the configuration. We can imagine the *create list with* block being replaced with a *parallel* block and each item attached being run in parallel with one another. However, we run into the issue that a parallel process can be arbitrarily large and be made up of many blocks, which must be contained within a scope *statement input* block. Due to the size of the *statement input* blocks, which would contain each process, being large and the setup to add additional processes being relatively convoluted, this approach is not ideal.

3.3.2 Separator Blocks

Another approach is to use standard block sequencing and introduce *separator* blocks which alter the implicit sequentiality of pi-calculus. In other words, the separator block would replace the standard *dot* sequence character to a corresponding separator, either composition or summation.

While conceptually straightforward, this approach introduces scope ambiguity such that explicit parentheses would have to be added to group parallel processes together, at which point we are writing pi-calculus syntax in as a 1:1 translation but in block form, and is something we aim to avoid.

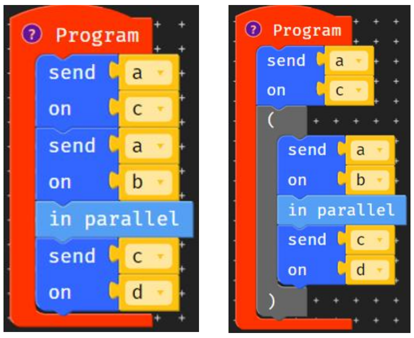


Figure 3.3: The parallel separator block and the explicit parenthesis (right) to avoid ambiguity with which send operations are in parallel with one another.

3.3.3 The List Pattern

The final approach that was considered fully leveraged the *statement input* of blocks to define scopes which alter the default sequencing behaviour. With this we can define two blocks, *run in parallel* and *choose from* for *composition* and *summation* respectively. *Run in parallel* runs all of the children blocks nested within it in parallel, while *choose from* makes each child block a choice. However, similarly to separator blocks, we need to define groups of blocks which should be part of a single parallel task or choice.

Therefore, we define two pairs of blocks which shall always be used together, *run in parallel* with *task* and *choose from* with *choice*, where *task* blocks may only ever exist within *run in parallel* blocks, while *choice* can only ever exist in *choose from*. We call this pattern of a specific parent block and child block the *list pattern*, which allows us to sequentially append the children blocks within a parent indefinitely, where the child cannot exist outside its defined parent block, while a parent block may not accept any other blocks but the defined child block.

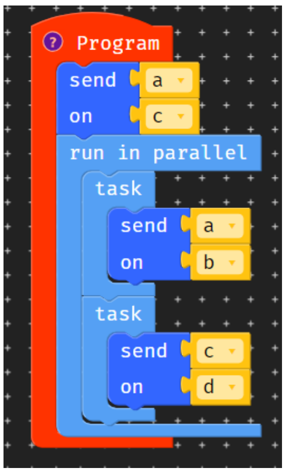


Figure 3.4: Parallel composition of two send actions.

This pattern of listing items within a parent introduced a way to reduce duplication, repetition and excessive nesting in the language. The problem of excessive nesting becomes evident when we are trying to restrict or receive many names in sequence within the same scope such as the following:

```
$a. $b. $c. $d. a(x). a(y). a(z). 0
```

Listing 3.3-1: Multiple name restricted or received in succession creating multiple scopes.

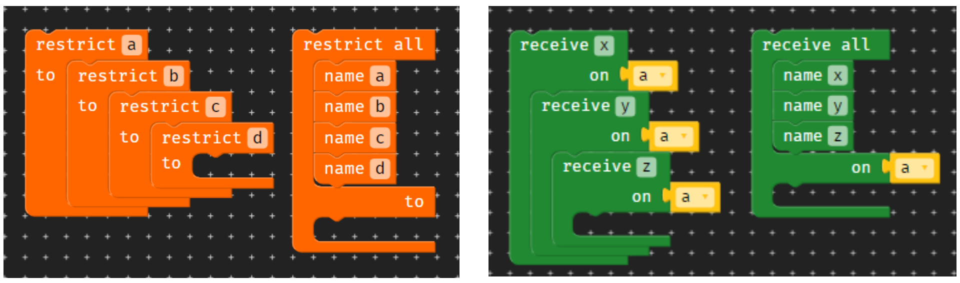


Figure 3.5: The list pattern used for restriction of multiple names at once (left) and receiving multiple names on the same channel at once (right). Corresponds to **Listing 3.3-1**.

4 Implementation

This section will cover the technical details of the implementation of the language and the web application features, including relevant code snippets.

4.1 Transpilation

Transpilation is the conversion from one programming language to another. In this case it is the conversion from blocks to PIFRA pi-calculus syntax.

4.1.1 Code Scrubbing

Code scrubbing refers to a function available in the Blockly API which is a function that runs in between the code generator functions of consecutive. In standard programming languages, this function can handle indentation and new lines, whereas for our purposes, we shall use it to determine the appropriate separator character depending on the blocks being evaluated.

```
const nextBlock = block.nextConnection?.targetBlock() || null;

const next = this.blockToCode(nextBlock);

const hasCode =
     typeof next === "object"
     ? next[0].replace(/\s/g, "").length > 0
     : next.replace(/\s/g, "").length > 0 && code.length > 0;

if (hasCode) {
     if (ChoiceScopes(block, nextBlock)) return code + " + " + next;
     if (ParallelScopes(block, nextBlock)) return code + " | " + next;
     if (ShouldSeparate(block) && nextBlock) return code + ". " + next;
}

return code + next;
```

Listing 4.1-1: Code Scrubbing function, determining the type of separator.

The function checks whether the next block of code has produced actual code, and if so it determines the separator to use based on the current and next block pair, with the default being no separator if all checks failed.

```
const ChoiceScopes = (a, b) => {
    return a?.type === "ChoiceScopeBlock" &&
           b?.type === "ChoiceScopeBlock";
};
```

Listing 4.1-2: Code scrubber predicate determining if two blocks are both Choice Blocks.

```
const ParallelScopes = (a, b) => {
    return a?.type === "ParallelScopeBlock" &&
           b?.type === "ParallelScopeBlock";
};
```

Listing 4.1-3: Code scrubber predicate determining if two blocks are both Task Blocks.

```
const ShouldSeparate = (block: Blockly.Block | null) => {
    switch (block?.type) {
        case "SendBlock":
        case "SyncBlock":
        case "MultiSendBlock":
        case "ReceiveScopeBlock":
        case "TerminationBlock":
            return true;
        default:
            return false;
    }
};
```

Listing 4.1-4: Code scrubber predicate determining if a dot sequence character should be inserted.

4.1.2 Code Style Enforcement

While PIFRA does not explicitly require that names or processes begin with an upper or lowercase letter or underscore specifically, the convention in pi-calculus is to keep names lowercase and process names uppercase. We shall enforce this convention as it makes it immediately obvious what the name is referring to.

To enforce this style we shall define two functions which verify that the user input conforms with the required convention, and if not it should correct the input to the best closest match.

```
const ToLowerleadingAlphaNumeric = (str: string) => {
    str = str.replace(/[^a-zA-Z0-9_]/g, "");
    str = str.replace(/^[0-9]+/, "");
    str = str.charAt(0).toLowerCase() + str.slice(1);
    return str;
};
```

Listing 4.1-5: Function enforcing a lowercase or underscore leading identifier.

```
const ToUpperleadingAlphaNumeric = (str: string) => {
    str = str.replace(/[^a-zA-Z0-9_]/g, "");
    str = str.replace(/^[0-9_]+/, "");
    str = str.charAt(0).toUpperCase() + str.slice(1);
    return str;
};
```

Listing 4.1-6: Function enforcing an uppercase leading identifier.

`ToLowerleadingAlphaNumeric` first removes all characters which are not valid for identifiers in PIFRA, then removes all leading numbers, as identifiers may not begin with a number, and finally sets the first character to lowercase. We can use this function to force names in the language to always be well-formed and begin with lowercase.

`ToUpperleadingAlphaNumeric` first removes all characters which are not valid for identifiers just like `ToLowerleadingAlphaNumeric`, it then removes all leading numbers and underscores and finally sets the first character of the remaining string to uppercase. This function can be used to force process names in the language to be well-formed identifiers which must begin with a capital letter.

4.2 Block Implementations

Each block is made up of 3 main components: *toolbox entry, block definition* and *code generation*. The toolbox entry is a declaration of the block and how it should appear in the toolbox of the block editor. It can contain defaults for the block for convenience. The block definition is the code used to determine the shape, colour and connections a block has. Finally, the code generation is a function which determines how each block generates code.
In addition to these components, some blocks contain additional unique behaviours which will be discussed where relevant.

A block is also assigned to one of 3 categories, *Scopes, Processes* and *Other*. The Scopes category includes all blocks which create scopes in the language or take other blocks as statement inputs. The Processes category contains all blocks used for the definition and the use of processes. Finally, the Other category includes all other blocks which do not warrant their own unique category.

Toolbox entries are written in JSON, but only the actual components related to each block will be shown, likewise, the instantiation, on-change and code generation functions will also be given as the bodies of the functions only for brevity.

4.2.1 Connection Restrictions

Certain blocks in the language have strict rules regarding which blocks they can connect to or which blocks can connect to them. If a block violates a rule it gets detached.

```
const MustBeInExactScope = (block: any, parent: string, scope: string) =>
{
    const surroundingParent = block.getSurroundParent();
    if (!surroundingParent) {
        if (block.getParent()?.type === parent) {
            block.unplug(false);
            return true;
        }
        return false;
    }

    if (surroundingParent.type !== parent ||
GetDirectChildren(surroundingParent, scope).indexOf(block) === -1) {
        block.unplug(false);
        return true;
    }
    return false;
};
```

Listing 4.2-1: Function forcing a block to be only ever attached to a specific scope of a specific parent block.

```
const CanOnlyContain = (block: any, scope: string, types: string[]) => {
    const children = GetDirectChildren(block, scope);
    const first = children.filter((c) => !types.includes(c.type))[0];
    if (first) {
        first.unplug(false);
        return true;
    }
    return false;
};
```

Listing 4.2-2: Function restricting the types of blocks are accepted within a specific scope of the current block.

```
const CanOnlyBeAttachedTo = (block: any, types: string[]) => {
    const prev = block.getParent();
    const next = block.getNextBlock();

    if (prev && !types.includes(prev.type)) {
        block.unplug(false);
        return true;
    }

    if (next && !types.includes(next.type)) {
        block.unplug(false);
        return true;
    }
    return false;
};
```

Listing 4.2-3: Function restricting what blocks the current block can attach to.

These functions will be referenced in the block definitions where applicable.

4.2.2 Global Name Block

The *global name* block is used to define the free names in the program. The purpose of this block is to inform the *Name Scope Manager*, which shall be discussed later, of the free names which we intend to have in the program.

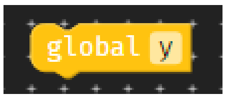


Figure 4.1: The global name block. Used to explicitly define free names.

4.2.2.1 Toolbox Entry

```
kind: 'block',
type: 'GlobalNameBlock'
```

Listing 4.2-4: Toolbox entry of the Global Name Block.

The entry of this block is empty and only contains the type of the block as there are no defaults we can provide for this block, as it is up to the user to input a free name of their choosing.

4.2.2.2 Block Definition

```
this.appendDummyInput()
     .appendField("global")
     .appendField(
         new Blockly.FieldTextInput("", ToLowerleadingAlphaNumeric), "NEW"
     );
this.setColour("#FFC310");
this.setNextStatement(true, null);
this.setPreviousStatement(true, null);
```

Listing 4.2-5: Global Name Block instantiation function.

The global name block is defined as a single text input. The text input makes use of the `ToLowerleadingAlphaNumeric` function which enforces that the name given begins with a lowercase letter or an underscore, and may include any letter, underscore or number thereafter. Note the "NEW" field name used in the text input field. This convention of the "NEW" field name is used across all blocks which introduce names into the program.

The block allows for a previous and next statement connection, meaning it can both attach to a block, and have other blocks being attached to it.
Finally, in its `onchange` function, it contains the guarding function `CanOnlyBeAttachedTo(this, ["GlobalNameBlock"])` which only allows it to be attached to other global name blocks, as we do not want this block to be inserted inside actual program code, but want to allow the block to be able to join together for cleanliness and organisation of blocks.

4.2.2.3 Code Generation

This block does not generate any code as free names are not explicitly indicated by the syntax of pi-calculus, and during the code generation pass only informs the name scope manager of the value it contains.

4.2.3 Name Accessor Block

The *name accessor* block is the only block in the language which is a *value input* block. The purpose of the block is for the user to select a valid name which is in scope at any given point in the program. The block determines the list of valid names by inspecting its position in the program and requesting the list of available names from the name scope manager. We define a single name accessing block to prevent unnecessary code duplication given that the logic to retrieve names is always the same.

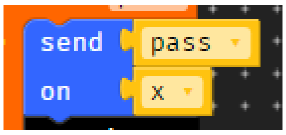


Figure 4.2: Two name accessor blocks attached to the send block.

4.2.3.1 Toolbox Entry

```
kind: "block",
type: "NameAccessBlock"
```

Listing 4.2-6: Toolbox entry of the Name Access Block.

This toolbox entry does not contain any defaults as this block is entirely dynamic in nature and the values it offers are dependent on the position of the block and the available names in the program in that position.

4.2.3.2 Block Definition

```
this.appendDummyInput().appendField(
     new Blockly.FieldDropdown([
          ["?", VISPI_INVALID_NAME],  ...NameAccessStates
     ]),
    "NAME"
);
this.setColour("#FFC310");
this.setOutput(true, null);
```

Listing 4.2-7: Name Access Block instantiation function.

Upon initialisation, the block is a dropdown list containing the default invalid name. In the case of loading the program from a file, we require that the loaded program's selected names persist, therefore we add the additional `NameAccessStates` which is a global constant which contains the loaded names. This is required as otherwise loading the dropdown without those additional names would cause data loss as all names would default back to the invalid name.

This is usually not an issue with other Blockly languages as dropdowns are often hard-coded, but given we can dynamically add and remove names, we must carefully manage the state so as to not accidentally remove the blocks data.

The update function of this block, which like other update functions, runs with every single change in the editor canvas, therefore if a name was added at some point and it became available to block, the dropdown must now reflect that therefore we must update the dropdown contents with each change.

```
const names = ScopeManager.GetLastScope()?
              .GetNames(GetAncestry(this.getParent())) ?? [];
```

Listing 4.2-8: Accessing names in-scope from the scope manager.

First we collect the names from the previous state of the program, before it was refreshed to know which names we had available right before the Blockly state refreshed. The exact reasons as to why we must use the previous state is discussed later in *Name Scope Manager*.

To determine which names we should provide, we are required to get the block's ancestry, which is a list of IDs of the blocks which preceded the current block. This allows us to determine within which scopes we are positioned in the program, and can provide the appropriate names.

```
this.getField("NAME").menuGenerator_ = [
    ["?", VISPI_INVALID_NAME], ...names.map((n) => [n, n])
];
```

Listing 4.2-9: Updating the available names of the Name Access Block.

We then update the dropdown menu by appending the names to the invalid name, which is always the first element in the list. We map over the names to produce tuples as Blockly requires the dropdown item to be in a format where the tuple is `[visiblevalue, internalvalue]`.

We must also consider a scenario where a name accessor block has been moved to a different scope where its name no longer exists. This is necessary to ensure that the intended correctness of the program is maintained. Instead of setting the value of the block to be the default invalid value, we instead display a warning to the user informing them of the fact that the name is not available in the current scope. Doing so we inform the user of the mistake, and by not changing the block's value, we prevent accidental actions of the user from being destructive.

```
if (!names.includes(value)) {
     this.setWarningText("Name is not in scope.");
} else {
     this.setWarningText(null);
}
```

Listing 4.2-10: Setting a warning message on the Name Access Block when the selected name is not in-scope.

4.2.3.3 Code Generation

During the code generation step, the block verifies if the name provided is the invalid default, if so, no code should be generated, otherwise we just return the name.

```
const name = block.getFieldValue("NAME");
if (name === VISPI_INVALID_NAME) return ["", 0];
return [name, 0];
```

Listing 4.2-11: Name Access Block code generation function.

4.2.4 Send Blocks

Send blocks refer to all blocks which compile down to the pi-calculus send syntax. There are three variations of the send block: *send, send all* and *sync*. Send is the standard send block which sends a single name on some other named channel. The send all block sends all specified names, through the use of the list pattern, on some named channel. Finally, the sync block sends a name across itself, which acts like a synchronisation or signalling that some action occurred, but no data was necessarily being sent.
All send blocks are part of the *Other* category.

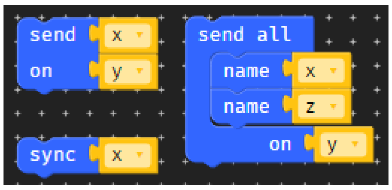


Figure 4.3: All send block variants, the standard send block (top left), the send all block (right) and the sync block (bottom left).

The code output from the blocks shown in **Figure 4.3** would be as follows:

Send: `y'<x>`

Send all: `y'<x>. y'<z>`

Sync: `x'<x>`

The send all and sync blocks were included for convenience due to commonly recurring patterns in pi-calculus which involve sending multiple names on the same channel or sending the channel on itself.

4.2.4.1 Toolbox Entry

Since all send blocks use defined names, and do not introduce any new names to the scope, they all by default have the *name access* block already attached to both the name that is being sent and the channel on which the message is being sent.

```
kind: "block",
type: "SendBlock",
inputs: {
     ON: {
          block: {
               type: "NameAccessBlock",
               fields: {
                    NAME: VISPI_INVALID_NAME,
               },
          },
     },
     MESSAGE: {
          block: {
               type: "NameAccessBlock",
               fields: {
                    NAME: VISPI_INVALID_NAME,
               },
          },
     },
}
```

Listing 4.2-12: Toolbox entry of the Send Block.

The default send block has both the message and the channel value inputs set to the name access blocks with the invalid name default value.

```
kind: "block",
type: "SyncBlock",
inputs: {
     ON: {
          block: {
               type: "NameAccessBlock",
               fields: {
                    NAME: VISPI_INVALID_NAME,
               },
          },
     },
}
```

Listing 4.2-13: Toolbox entry of the Sync Block.

The sync block is identical to the default send block apart from the fact that the message and channel are the same, hence we only require one field to be set with the default name access block value. The send all block is different to the previous two blocks given it uses the list pattern, in which it is the parent block. Therefore we require the child block to be defined which will be used to list an unlimited amount of names to be sent.

```
kind: "block",
type: "SendNameBlock",
inputs: {
     MESSAGE: {
          block: {
               type: "NameAccessBlock",
               fields: {
                    NAME: VISPI_INVALID_NAME,
               },
          },
     },
}
```

Listing 4.2-14: Toolbox entry of the Send Name Block. (Child block of Send All Block)

This block simply holds onto a name access block and is a child block of the send all block.

```
kind: "block",
type: "MultiSendBlock",
inputs: {
     ON: {
          block: {
               type: "NameAccessBlock",
               fields: {
                    NAME: VISPI_INVALID_NAME,
               },
          },
     },
     MESSAGES: {
          block: {
               type: "SendNameBlock",
               inputs: {
                    MESSAGE: {
                         block: {
                              type: "NameAccessBlock",
                              fields: {
                                   NAME: VISPI_INVALID_NAME,
                              },
                         },
                    },
               },
          },
     },
}
```

Listing 4.2-15: Toolbox entry of the Send All Block.

The send all block comes with a default name access block for the channel name on which the messages are being sent, and a single *send name child* block, as at least one name must be provided.

4.2.4.2 Block Definitions

The default send block does not require any updates as it itself does not access the name data from the scope manager, and the block is allowed to be attached anywhere in the program, unless the parent block it is attaching to disallows it.

```
this.appendValueInput("MESSAGE").appendField("send");
this.appendValueInput("ON").appendField("on");
this.setPreviousStatement(true, null);
this.setNextStatement(true, null);
this.setColour("#3366FF");
```

Listing 4.2-16: Send Block instantiation function.

We simply append two value input fields, for the channel and message names. We also allow for the block to attach to other blocks, and other blocks to attach to itself by allowing the previous and next statements.

The sync block follows suit in a similar manner, and is in fact simpler since it requires just one name.

```
this.appendValueInput("ON").appendField("sync");
this.setPreviousStatement(true, null);
this.setNextStatement(true, null);
this.setColour("#3366FF");
```

Listing 4.2-17: Sync Block instantiation function.

The initialisation definition for the send all child block is identical to that of the sync block, however we restrict the block's attachment so that it can only exist within its parent block.

```
if (CanOnlyBeAttachedTo(this, ["MultiSendBlock","SendNameBlock"])) return;
MustBeInExactScope(this, "MultiSendBlock", "MESSAGES");
```

Listing 4.2-18: Send Name Block onchange function, restricting allowed block connections.

If the `CanOnlyBeAttachedTo` function finds that the block is attached to something other than specified, it will detach the block and exit early. If the block was not detached, meaning it is attached to the allowed blocks, we must check that it is within the valid scope inside the send all block, and is not attached to the bottom of the block in a sequence.

Finally, the send block itself contains a statement input for the children blocks, and the value input for the channel name.

```
this.appendDummyInput().appendField("send all")
     .setAlign(Blockly.inputs.Align.RIGHT);
this.appendStatementInput("MESSAGES");
this.appendValueInput("ON").appendField("on")
     .setAlign(Blockly.inputs.Align.RIGHT);
this.setPreviousStatement(true, null);
this.setNextStatement(true, null);
this.setColour("#3366FF");
```

Listing 4.2-19: Send All Block instantiation function.

The final requirement for this block is to disallow blocks other than the specific child block from being attached within its statement input, we do this by detaching blocks which do not match the list of allowed blocks.

```
CanOnlyContain(this, "MESSAGES", ["SendNameBlock"]);
```

Listing 4.2-20: Send All Block restriction on allowed children blocks in its scope.

4.2.4.3 Code Generation

To generate the code for send blocks, we first retrieve the values generated by the name accessor blocks, after which we verify that all the values are valid.

```
const name = generator.valueToCode(block, "ON", 0);
const value = generator.valueToCode(block, "MESSAGE", 0);

if (name.length > 0 && value.length > 0) {
    return `${name}'<${value}>`;
}

return ``;
```

Listing 4.2-21: Send Block code generation function.

For *send name* blocks, the children blocks of *send all* blocks, we do not generate any code, instead we delegate the generation of the code to the send all block.

```
const on = generator.valueToCode(block, "ON", 0);
const messages = GetDirectChildren(block, "MESSAGES")
         .map((message) => generator.valueToCode(message, "MESSAGE", 0))
         .filter((m) => m.length > 0);

if (on.length > 0 && messages.length > 0) {
    return messages.map((m) => `${on}'<${m}>`).join(". ");
}
return ``;
```

Listing 4.2-22: Send All Block code generation function.

We map over all the children blocks and generate their values, after which we filter out any empty names. Then, if we have a valid name used as the channel and at least one name to be sent, we map over the names and join them with the sequence dot delimiter. We must join the send actions manually since this code does not go through the code scrubber.

4.2.5 Receive Blocks

Receive blocks are scoping blocks since they introduce names to a scope, and as such are part of the *Scopes* category.

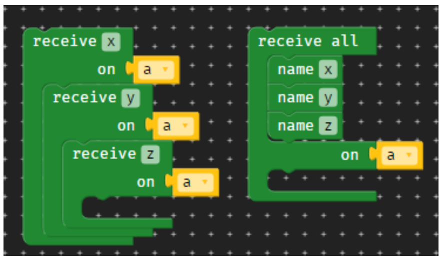


Figure 4.4: Two variants of the receive block. The standard receive block allows for receiving a single name (left) and the receive all block (right) allowing for receiving multiple names at a time. Left and right representations are equivalent.

4.2.5.1 Toolbox Entry

The standard receive block is defined with a default name accessor block for the channel on which a name would be received.

```
kind: "block",
type: "ReceiveScopeBlock",
inputs: {
    ON: {
        block: {
            type: "NameAccessBlock",
            fields: {
                NAME: VISPI_INVALID_NAME,
            },
        },
    },
}
```

Listing 4.2-23: Toolbox entry of the Receive Block.

For the receive all block we follow the exact same pattern as in the send all block, where we include a default *receive name* child block.

```
kind: "block",
type: "ReceiveNameBlock"
```

Listing 4.2-24: Toolbox entry of the Receive Name Block. (Child block of Receive All Block)

The receive name block does not contain any defaults itself as it is a text input block, and the user is required to enter a name they wish to use.

```
kind: "block",
type: "MultiReceiveScopeBlock",
inputs: {
    NAMES: {
        block: {
            type: "ReceiveNameBlock",
        },
    },
    ON: {
        block: {
            type: "NameAccessBlock",
            fields: {
                NAME: VISPI_INVALID_NAME,
            },
        },
    },
}
```

Listing 4.2-25: Toolbox entry of the Receive All Block.

The receive all block is defined exactly in the same way we defined the *send all* block, providing a default empty *receive name* block and a default *name accessor* block.

4.2.5.2 Block Definitions

The receive block contains a single text input field and a statement input field. It is the first block to not allow any other blocks from being attached to it as once a name is defined, it is available for the remainder of the scope, hence all consecutive blocks must be inserted into the body of the block.

```
this.appendDummyInput()
     .appendField("receive")
     .appendField(
          new Blockly.FieldTextInput("", ToLowerleadingAlphaNumeric), "NEW"
     );

this.appendValueInput("ON")
     .appendField("on").setAlign(Blockly.inputs.Align.RIGHT);

this.appendStatementInput("SCOPE");
this.setColour("#208932");
this.setPreviousStatement(true, null);
this.setNextStatement(false, null);
```

Listing 4.2-26: Receive Block instantiation function.

Similarly to the send all block, the receive all block requires a child block defined for itself, given it uses the list pattern.

```
this.appendDummyInput()
     .appendField("name")
     .appendField(
          new Blockly.FieldTextInput("", ToLowerleadingAlphaNumeric), "NEW"
     );
this.setColour("#208932");
this.setPreviousStatement(true, null);
this.setNextStatement(true, null);
```

Listing 4.2-27: Receive Name Block instantiation function. (Child block of Receive All Block)

```
if (CanOnlyBeAttachedTo(this,
     ["MultiReceiveScopeBlock","ReceiveNameBlock"])
) return;
MustBeInExactScope(this, "MultiReceiveScopeBlock", "NAMES");
```

Listing 4.2-28: Receive Name Block onchange function, restricting allowed block connections.

The receive all block adds an additional scope for listing names to receive, just like the send all block adds a scope for listing the names to send.

Additionally, it also follows the same pattern in restricting the allowed blocks which may be inserted into the name scope.

```
this.appendDummyInput()
     .appendField("receive all")
     .setAlign(Blockly.inputs.Align.RIGHT);

this.appendStatementInput("NAMES");
this.appendValueInput("ON")
     .appendField("on")
     .setAlign(Blockly.inputs.Align.RIGHT);

this.appendStatementInput("SCOPE");
this.setColour("#208932");
this.setPreviousStatement(true, null);
this.setNextStatement(false, null);
```

Listing 4.2-29: Receive All Block instantiation function.

```
CanOnlyContain(this, "NAMES", ["ReceiveNameBlock"]);
```

Listing 4.2-30: Receive All Block restriction on allowed children blocks in its scope.

4.2.5.3 Code Generation

```
const name = block.getFieldValue("NEW");
const on = generator.valueToCode(block, "ON", 0);
ScopeManager.RegisterBlockScope(block);
ScopeManager.InsertName(name);
const scope = generator.statementToCode(block, "SCOPE");
ScopeManager.PopScope();

if (name.length > 0 && on.length > 0) {
     if (scope.length > 0) return `${on}(${name}). ${scope}`;
     else return `${on}(${name})`;
}

return ``;
```

Listing 4.2-31: Receive Block code generation function.

The receive block retrieves both the received name and channel name values and proceeds to invoke the `RegisterBlockScope` function with the *Scope Manager*. This is the first time we define a scope and from this point onwards, any `InsertName` function call will insert a name into the active scope, until a new scope is registered or the current scope is popped off the scope stack. The `generator.statementToCode` call traverses the Blockly AST to generate the code for that statement, which will have the currently registered scope being active. Finally, if all the values are present we generate the code, else we do not generate any code until the receive block is completely well-formed by the user.

```
const name = block.getFieldValue("NEW");

if (name.length > 0) {
     ScopeManager.InsertName(name);
}
return "";
```

Listing 4.2-32: Receive Name Block code generation function.

Unlike the send name block, the receive name block does have an implementation which is solely responsible for inserting the names into the current scope, but does not on its own generate any code.

```
ScopeManager.RegisterBlockScope(block);
const names = GetDirectChildren(block, "NAMES")
     .map((name) => name.getFieldValue("NEW"))
     .filter((n) => n.length > 0);

const on = generator.valueToCode(block, "ON", 0);
generator.statementToCode(block, "NAMES");

const scope = generator.statementToCode(block, "SCOPE");
ScopeManager.PopScope();

if (scope.length > 0) {
     return `${names.map((n) => `${on}(${n})`).join(". ")}. ${scope}`;
}

return `${names.map((n) => `${on}(${n})`).join(". ")}`;
```

Listing 4.2-33: Receive All Block code generation function.

Given that the receive name blocks insert the names, we begin the receive all block by registering the scope. We gather the names and the channel on which we will be receiving them, then proceed to invoke the `generator.statementToCode` to insert the names into the scope.

4.2.6 Restrict Blocks

The restrict blocks act in a very similar manner to the receive blocks with the main difference being that they do not require a name accessor block at all. These blocks are also part of the *Scopes* category.

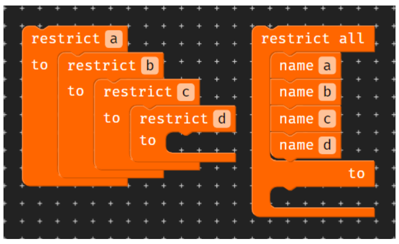


Figure 4.5: Standard Restrict Blocks (left) and the Restrict All Block (right). Left and right representations are equivalent.

4.2.6.1 Toolbox Entry

Given that the restrict blocks are mainly text inputs for which we cannot provide default values, the toolbox entries are straightforward.

```
kind: "block",
type: "RestrictScopeBlock"
```

Listing 4.2-34: Toolbox entry of the Restrict Block.

```
kind: "block",
type: "RestrictNameBlock"
```

Listing 4.2-35: Toolbox entry of the Restrict Name Block. (Child block of the Restrict All Block)

```
kind: "block",
type: "MultiRestrictScopeBlock",
inputs: {
     NAMES: {
          block: {
               type: "RestrictNameBlock",
          },
     },
}
```

Listing 4.2-36: Toolbox entry of the Restrict All Block.

The restrict all block is the only restrict block variation which has a default block defined which is an empty restrict name block, as we always need at least one for the block to be well-formed.

4.2.6.2 Block Definitions

The block definitions of restrict blocks are nearly identical to the receive blocks, with the main differences being style, labelling and the lack of the name access block value input.

The *restrict* block is defined exactly as the *receive* block in **Listing 4.2-26** with the only difference being the lack of the value input field.

The *restrict name* block is defined exactly as the *receive name* block in **Listing 4.2-27** and its connection restrictions are defined in the same manner as seen in **Listing 4.2-28** with all mentions of the *receive* block types being replaced by *restrict* block type equivalents.

The *restrict all* block is also defined exactly as seen for the *receive all* block in **Listing 4.2-29** and its connection restriction is done as seen in **Listing 4.2-30** with the difference being the use of the *restrict name* block instead of the *receive name* block.

4.2.6.3 Code Generation

```
const name = block.getFieldValue("NEW");
if (name === VISPI_INVALID_NAME || name.length === 0) return "";

ScopeManager.RegisterBlockScope(block);
ScopeManager.InsertName(name);
const scope = generator.statementToCode(block, "SCOPE");
ScopeManager.PopScope();

if (scope.length > 0) {
    return `$${name}. ${scope}`;
}

return `$${name}`;
```

Listing 4.2-37: Restrict Block code generation function.

The restrict block's code generation function follows the pattern seen in **Listing 4.2-31** with the difference being the PIFRA code output and the lack of the value input from the name accessor block which does not exist on restrict blocks.

The code generation for the *restrict name* and *restrict all* blocks follows the pattern seen **Listing 4.2-32** and **Listing 4.2-33** respectively, with the differences being the same as mentioned above.

4.2.7 Termination Block

The termination block is the simplest block in the entire language as it has no user input and always generates the same code, that being the termination character `0`. In addition to that, the block is allowed to be attached to any viable blocks, but it itself does not allow any other blocks to follow.

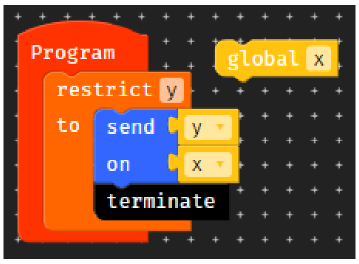


Figure 4.6: The termination block used to define the termination of the program.

4.2.8 Parallel Blocks

The parallel blocks are a list pattern pair of child-parent blocks, the *run in parallel* parent block and the *task* child block.

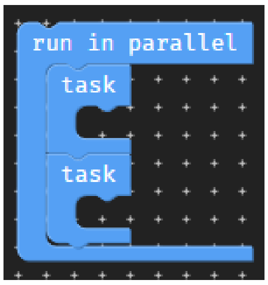


Figure 4.7: The Parallel Block and two Task Blocks.

4.2.8.1 Toolbox Entry

```
kind: "block",
type: "ParallelParentBlock",
inputs: {
     PARALLEL: {
          block: {
               type: "ParallelScopeBlock",
          },
     },
}
```

Listing 4.2-38: Toolbox entry for the Parallel Block.

The parallel block is defined to always have a single task block attached to it as it cannot ever hold any other blocks directly within its scope, as those must be placed in a task block instead.

```
kind: "block",
type: "ParallelScopeBlock"
```

Listing 4.2-39: Toolbox entry for the Task Block. (Child block of the Parallel Block)

The task block itself does not use any defaults as its main purpose is to divide sections of processes into units which run in parallel within the context of its parallel block parent.

4.2.8.2 Block Definition

```
this.appendDummyInput().appendField("run in parallel");
this.appendStatementInput("PARALLEL").setCheck("ParallelScopeBlock");
this.setPreviousStatement(true, null);
this.setNextStatement(false, null);
this.setColour("#55A0F4");
```

Listing 4.2-40: Parallel Block instantiation function.

The parallel block is a simple block made up of only a single statement input which is only allowed to hold task blocks. The block does not allow any other blocks to be attached to the end of it as all parallel processes must terminate to indicate the termination of the entire parallel composition.

Similarly to prior list pattern parent blocks, the parallel block will only allow a specific child block to be accepted within its scope.

```
CanOnlyContain(this, "PARALLEL", ["ParallelScopeBlock"]);
```

Listing 4.2-41: Parallel block restriction on allowed children blocks.

The task block is also a simple block, just like the parallel block with the only difference is that other blocks can be attached to it, specifically only other task blocks.

```
this.appendDummyInput().appendField("task");
this.appendStatementInput("SCOPE");
this.setPreviousStatement(true, null);
this.setNextStatement(true, "ParallelScopeBlock");
this.setColour("#55A0F4");
```

Listing 4.2-42: Task Block instantiation function.

Similarly to other list pattern child blocks, the task block restricts itself to only be attachable within the scope of its parent.

```
MustBeInExactScope(this, "ParallelParentBlock", "PARALLEL");
```

Listing 4.2-43: Task block connection restriction to the scope of its parent block.

4.2.8.3 Code Generation

```
const parallelCount = GetDirectChildren(block, "PARALLEL")
     .filter(
        (b) => generator.statementToCode(b, "SCOPE")
              .replace(/\s/g, "").length > 0
     ).length;
ScopeManager.RegisterBlockScope(block);
const scopes = generator.statementToCode(block, "PARALLEL");
ScopeManager.PopScope();
if (parallelCount > 1) {
     if (["MainBlock","ProcessBlock"]
          .includes(ScopeManager.GetCurrentScopeType()))
          return scopes;
     return `(${scopes})`;
} else if (parallelCount === 1) return scopes;
else return "";
```

Listing 4.2-44: Parallel Block code generation function.

The parallel block code generation function acts similar to other scoping blocks in that it registers a scope and proceeds to evaluate and generate the code for the contents of its scope. We also check for whether at least two task blocks are found within the parallel block, as this indicates that we may need to place parentheses around the generated code if the parallel block does not lie directly within a main block or process block scope.

```
ScopeManager.RegisterBlockScope(block);
const scope = generator.statementToCode(block, "SCOPE");
ScopeManager.PopScope();
if (scope.replace(/\s/g, "").length === 0) return "";
return scope;
```

Listing 4.2-45: Task Block code generation function.

The task block simply registers a scope, evaluates its contents and if the contents are not empty or only whitespace, it returns the generated code.

4.2.9 Choice Blocks

The choice blocks, just like parallel blocks, are a pair of blocks which form a list pattern with the *choose* block and *choice* block being the parent and child blocks respectively.

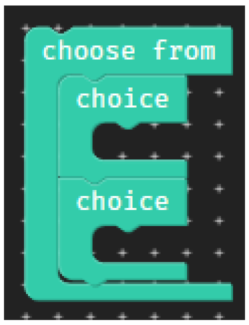


Figure 4.8: The Choose Block and two Choice Blocks.

The purpose of the choice blocks is to create separate processes from which only one is to execute, just like the task blocks separate processes into parallel tasks.

4.2.9.1 Toolbox Entry

The choose block follows the same structure and defaults as the parallel block as seen in **Listing 4.2-38**, while the choice block follows the same structure as the task block in **Listing 4.2-39**.

4.2.9.2 Block Definitions

Just like with the toolbox entries, the block definitions and connection restrictions for the choose and choice blocks follow the same patterns as the parallel and task blocks as seen in **Listing 4.2-40**, **Listing 4.2-41**, **Listing 4.2-42** and **Listing 4.2-43**.

4.2.9.3 Code Generation

Given the choose and choice blocks follow the exact same patterns as the parallel and task blocks, their code generation functions are identical to **Listing 4.2-44** and **Listing 4.2-45** respectively, as the separator of choices is handled by the code scrubber function.

4.2.10 Guard Block

Guard blocks are akin to *if statements* in other block-based programming languages with the difference being they can only be an equality or inequality and cannot have an else statement following them.

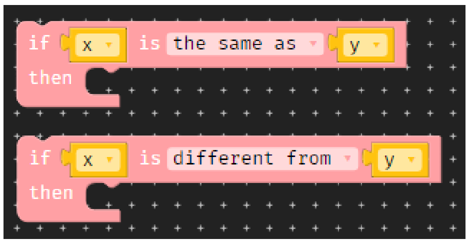


Figure 4.9: An equality Guard Block (top) and an inequality Guard Block (bottom).

4.2.10.1 Toolbox Entry

```
kind: "block",
type: "GuardScopeBlock",
inputs: {
     FIRST: {
          block: {
               type: "NameAccessBlock",
               fields: {
                    NAME: VISPI_INVALID_NAME,
               },
          },
     },
     SECOND: {
          block: {
               type: "NameAccessBlock",
               fields: {
                    NAME: VISPI_INVALID_NAME,
               },
          },
     },
}
```

Listing 4.2-46: Toolbox entry for the Guard Block.

The guard block is defined to by default contain two name accessor blocks which are used in the equality and inequality checks.

4.2.10.2 Block Definition

```
const GuardOperators: Blockly.MenuGenerator = [
     ["is the same as", "="],
     ["is different from", "!="],
];

this.appendValueInput("FIRST").appendField("if");
this.appendDummyInput().appendField(
     new Blockly.FieldDropdown(GuardOperators),
     "OPERATION"
);
this.appendValueInput("SECOND");
this.appendStatementInput("SCOPE").appendField("then");
this.setColour("#FFA0A4");
this.setPreviousStatement(true, null);
this.setNextStatement(false, null);
```

Listing 4.2-47: Guard Block instantiation function.

The guard block is set to contain two value inputs for the name accessor blocks as well a dropdown which allows the block to flips between equality and inequality. Since nothing can be placed behind a guard block which is not bound by the guard check, the next statement input is disabled, making attaching other blocks to the end of this block impossible.

4.2.10.3 Code Generation

```
const first = generator.valueToCode(block, "FIRST", 0);
const operation = block.getFieldValue("OPERATION");
const second = generator.valueToCode(block, "SECOND", 0);

ScopeManager.RegisterBlockScope(block);
const scope = generator.statementToCode(block, "SCOPE");
ScopeManager.PopScope();

if (first.length > 0 && second.length > 0) {
     return `[${first}${operation}${second}] ${scope}`;
}

return ``;
```

Listing 4.2-48: Guard Block code generation function.

The guard block code generation function simply reads the values passed in as value inputs for the names and the comparison operation, registers its scope, and evaluates the scope.

4.2.11 Process Blocks

The process blocks refer to all blocks which are used for the definition of processes and their invocation, and are all part of the *Process* category.
The blocks used for processes are *process definition, process call, parameter* and *argument* blocks.
These four blocks form list pattern child-parent pairs, where the parameter block is the child of the process definition block and the argument block is the child of the process call block.

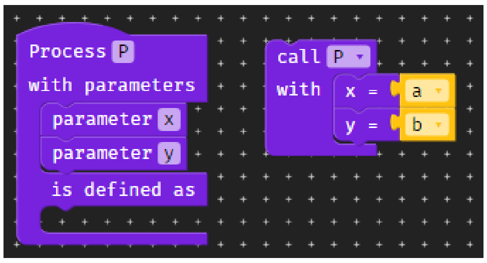


Figure 4.10: Process definition and parameter blocks (left) and the process call and argument blocks (right).

4.2.11.1 Toolbox Entries

```
kind: "block",
type: "ProcessBlock"
```

Listing 4.2-49: Toolbox entry for the Process Definition Block.

```
kind: "block",
type: "ProcessParamBlock"
```

Listing 4.2-50: Toolbox entry for the Process Parameter Block.

```
kind: "block",
type: "ProcessCallBlock"
```

Listing 4.2-51: Toolbox entry for the Process Call Block.

```
kind: "block",
type: "ProcessArgBlock",
inputs: {
    ARG: {
        block: {
            type: "NameAccessBlock",
            fields: {
                NAME: VISPI_INVALID_NAME,
            },
        },
    },
}
```

Listing 4.2-52: Toolbox entry for the Process Argument Block.

The process definition, parameter and call blocks do not have any defaults as a process may not require any parameters, and a process call may not require any arguments. Since the only valid input into a process call is a name, the argument block comes with a default name accessor block attached.

4.2.11.2 Block Definitions

```
const input = new Blockly.FieldTextInput(
    "UnnamedProcess",
    ToValidProcessName(this.id)
);
this.appendDummyInput()
    .appendField("Process").appendField(input, "PROCESS");
this.appendDummyInput()
    .appendField("with parameters").setAlign(Blockly.inputs.Align.RIGHT);
this.appendStatementInput("PARAMS");
this.appendDummyInput()
    .appendField("is defined as").setAlign(Blockly.inputs.Align.RIGHT);
this.appendStatementInput("BODY");
this.setColour("#7722DD");
this.setPreviousStatement(false, null);
this.setNextStatement(false, null);
```

Listing 4.2-53: Process Definition Block instantiation function.

The shape of the process definition block is a text input field for the process name, a scope for defining parameters using the list pattern and finally a scope for the body of the process. The default name a process has is an *UnnamedProcess*.
Since processes cannot be shadowed like names, we must carefully handle the names which are given to processes to ensure they do not clash, for which we use the `ToValidProcessName` function.

```
const ToValidProcessName = (blockid: string) => (str: string) => {
    str = ToUpperleadingAlphaNumeric(str);
    let scope = ScopeManager.GetLastScope();

    if (scope) {
        const id = scope.GetRawProcessId(str);
        if (id === undefined) return str;
        if (id !== blockid) {
            let i = 0;
            while (scope.GetRawProcessId(str + i) !== undefined) {
                i++;
            }
            str = str + i;
        }
    }

    return str;
}
```

Listing 4.2-54: Function enforcing valid process names.

The user provided name is first converted to a valid, uppercase process name, as per the style enforcement in VisPi.
Then we are required to verify whether a process of the given name already exists, if not, we can return the name as is, otherwise we verify that the process ID is not the same as the current block. This is required for deserialization so that if the block ID matches the process name, it should be retained, as we want to retain the exact naming which was used before the serialization. If the block ID does not match the block ID for the process name, it means that the block was either copied and pasted or a new process block was given the same name as an existing block. If this is detected, we append the name with an enumeration until a name which is free is found.

Since the process definition block uses the list pattern in which it is the parent block, it restricts the allowed children types within its parameter scope to be the parameter block only.

```
CanOnlyContain(this, "PARAMS", ["ProcessParamBlock"]);
```

Listing 4.2-55: Process Definition Block restriction on allowed children blocks in its *params* scope.

The process parameter block follows the exact same pattern established by receive and restrict name blocks (child blocks) for its instantiation function and connection restrictions, as seen in **Listing 4.2-27** and **Listing 4.2-28**.

```
this.appendDummyInput()
     .appendField("call")
     .appendField(new Blockly.FieldDropdown(ProcessNames), "PROCESS_NAME");
this.appendStatementInput("ARGS").appendField("with");
this.setColour("#7722DD");
this.setPreviousStatement(true, null);
this.setNextStatement(false, null);
```

Listing 4.2-56: Process Call Block instantiation function.

The process call block acts in a similar fashion to the name accessor block, as it requires a dynamic list of process names from which we are allowed to select from. The variable `ProcessNames` is a global variable instantiated during deserialization of a VisPi program, to ensure that the process names are recovered correctly.

```
CanOnlyContain(this, "ARGS", ["ProcessArgBlock"]);

let names = ScopeManager.GetLastScope()?.GetProcessNames();
const menu = names?.map((n) => [n, n]) ?? [["?", VISPI_INVALID_NAME]];

this.getField("PROCESS_NAME").menuGenerator_ = menu;

const paramCount = ScopeManager.GetLastScope()?
     .GetParams(this.getFieldValue("PROCESS_NAME")).length ?? 0;
const argCount = GetDirectChildren(this, "ARGS").length;

if (paramCount !== argCount) {
     this.setWarningText(
          `Expected ${paramCount} argument${paramCount === 1 ? "" : "s"},
           but got ${argCount}.`
     );
} else {
     this.setWarningText(null);
}
```

Listing 4.2-57: Process Call onchange function.

The function enforces that only process argument blocks are allowed to be inserted within the scope of the process call block.
Every time the program updates and potentially a new process was introduced into the program, we must update the dropdown menu of the process call block to ensure that all processes are available. Finally, the function verifies that the number of arguments provided to the block matches the defined number of parameters, otherwise we set a warning.

The process argument block instantiation function follows the same pattern as defined by the sync block and the send name block as seen in **Listing 4.2-17**.

```
const parentName = this.getSurroundParent()?
     .getFieldValue("PROCESS_NAME");
if (parentName) {
     const scope = ScopeManager.GetLastScope();
     let scopeNames = scope?.GetParams(parentName) ?? [];

     // When loading for the first time from JSON
     if (scopeNames.length === 0)
          scopeNames = ScopeManager.GetParams(parentName);

     const thisIndex = GetDirectChildren(this.getSurroundParent(), "ARGS")
               .indexOf(this);
     if (thisIndex < scopeNames.length && thisIndex !== -1) {
          this.setFieldValue(scopeNames[thisIndex] + " =", "LABEL");
          this.setWarningText(null);
     } else if (thisIndex >= scopeNames.length) {
          this.setWarningText(
               `Unexpected argument. Process '${parentName}' expects
               {scopeNames.length} arguments.`
          );
          this.setFieldValue("argument");
     }
}

if (CanOnlyBeAttachedTo(this, ["ProcessCallBlock", "ProcessArgBlock"]))
     return;
MustBeInExactScope(this, "ProcessCallBlock", "ARGS");
```

Listing 4.2-58: Process Argument Block onchange function.

The argument is a special block in that it changes its label inside the `onchange` function to match the parameter it is binding to. This allows the user to immediately see which value is being assigned to when the process is being invoked.

Similarly to the process call block, the argument block also displays a warning message if it exceeds the parameter count.

Finally, since it's a child block of the process call block, it restricts its connections to other instances of itself and the process call block.

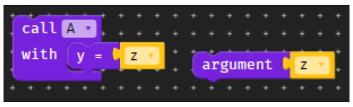


Figure 4.11: An argument block indicating the corresponding parameter it is binding to (left) and a default, unbound argument block (right).

4.2.11.3 Code Generation

```
ScopeManager.SetGeneration(true);

const process = block.getFieldValue("PROCESS");
const params = GetDirectChildren(block, "PARAMS");
let paramNames = [];
for (const param of params) {
     const name = param.getFieldValue("NEW");
     if (name) paramNames.push(name);
}

ScopeManager.RegisterProcess(block.id, process, paramNames);
const scope = generator.statementToCode(block, "BODY");
ScopeManager.PopScope();

let paramString = paramNames.join(", ");
if (paramString.length > 0) {
     paramString = `(${paramString})`;
}

ScopeManager.SetGeneration(false);
if (scope.replace(/\s/g, "").length === 0) return "";
return `${process}${paramString} = ${scope}`;
```

Listing 4.2-59: Process Definition Block code generation function.

The process block is the first of two blocks which make use of the Scope Manager's `SetGeneration` function. This tells VisPi that when the code generation is set to true it is allowed to generate code, otherwise it simply returns empty strings from all blocks. This prevents random floating blocks which are not scoped within a process from generating code in the output, which would result in syntactically invalid PIFRA code.

The function proceeds to gather the defined parameters and registers the process, which tells the scope manager about the existence of the process, its name and its parameters, and at the same time it defines a scope for the process and inserts the parameter names into the scope. We then proceed to generate the code for the entire process definition.

When generating the process parameter list, we check whether any parameters were provided, as we must enclose the parameter list with parentheses as per the syntax.
When the process block completes its code generation, it again disables any extra code from being generated until enabled again.

The process parameter and argument blocks do not produce any code, as their values are directly extracted and used by their parent blocks instead, just like the send name child block.

```
const process = block.getFieldValue("PROCESS_NAME");

if (process === VISPI_INVALID_NAME) return "";

const args = GetDirectChildren(block, "ARGS");
let argNames = [];
for (const arg of args) {
     const name = generator.valueToCode(arg, "ARG", 0);
     if (name) argNames.push(name);
}

let argString = argNames.join(", ");
if (argString.length > 0) {
     argString = `(${argString})`;
}

return `${process}${argString}`;
```

Listing 4.2-60: Process Call Block code generation function.

The process call block code generation is similar to the actual code generation component of the process definition block, where we simply collect and evaluate the arguments, verify whether there is at least one argument and if so wrap the arguments in parentheses, and output the process name with the arguments. However, if a valid process is not selected, no code will be generated.

4.2.12 Main Program Block

The main program block is a simple scope block, of which only one can ever exist in a VisPi program. Its purpose is to contain the system definition the user is modelling. This is the second block which enables code generation and its code generation function is simply evaluating the code of the blocks stored within its statement input.

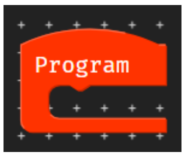


Figure 4.12: The main Program Block.

4.3 Name Scope Management

In Blockly [3], variable scoping is not provided by default, and languages generally set all variables as global. However, for the purposes of making learning pi-calculus more approachable with VisPi, we aim to help the user know which names they're allowed to use, which is what the scope manager was designed to do.

The scope management system aims to reduce the possibility of errors in VisPi programs to a minimum, by ensuring that the user can only ever access names which are in-scope at any given point in their program. By doing so, we can ensure that bound names declared out of a scope are not treated as free names in that scope by accident, which is a possibility when writing programs without any static analysis on the availability of names as well as the additional possibility of typo induced errors.

4.3.1 Scope Management System

When we generate code from blocks using the Blockly API, we traverse the AST created by Blockly from the blocks. This process follows that of any parser for any programming language, where we can perform certain processing by entering certain AST nodes. We use this to our advantage to read any name definitions defined in the nodes by the blocks, and use blocks which are scoping in nature to push their scope onto the stack of scopes. This system allows us to confidently gather the names which are in-scope by traversing up the stack, and also introduce the concept of name shadowing, which is the act of reusing the same identifier for a different purpose in a more deeply nested scope.

The scope manager is responsible for the following aspects of the program: name scopes, process definitions, storing whether a block is allowed to generate code, and whether the main program block has been declared already.

Example H: Name Scoping

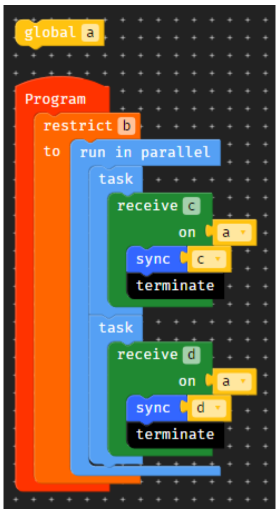

```
global a
Program
  restrict b
  to run in parallel
    task
      receive c
        on a
        sync c
        terminate
    task
      receive d
        on a
        sync d
        terminate
```

Figure 4.13: Example VisPi program with multiple scopes.

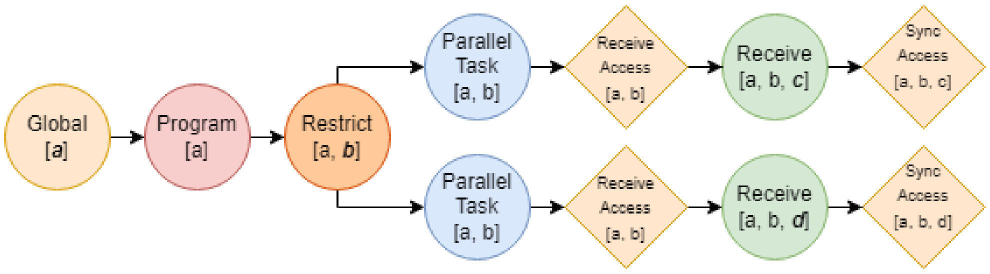
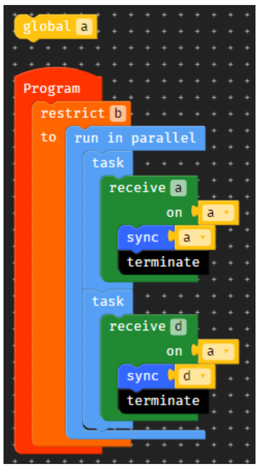


Figure 4.14: Graph of scopes and available names of the program in **Figure 4.13**.

In **Figure 4.14** we can see scopes defined with circle nodes, the list of available names denoted by brackets, name accesses denoted by diamonds and newly added names highlighted in bold italics.

From the graph we can see how VisPi handles name accesses by looking at the *Receive Access* nodes, which are the name access blocks attached to the receive blocks. These access nodes do not yet possess knowledge of the newly received names *c* and *d*. In the final access nodes in both paths we see the list of names that are available for selection. These names can be obtained by traversing the stack backwards towards the global scope and collecting the names defined along the way. This way each access block can create a *path* which leads to itself starting at the global scope. This creates a unique identifier which can be used to retrieve all names that should be available at the block.

Example I: Name Shadowing



```
global a

Program
  restrict b
  to run in parallel
    task
      receive a
        on a
      sync a
      terminate
    task
      receive d
        on a
      sync d
      terminate
```

Figure 4.15: Example VisPi program with the shadowing of a free name by a bound name.

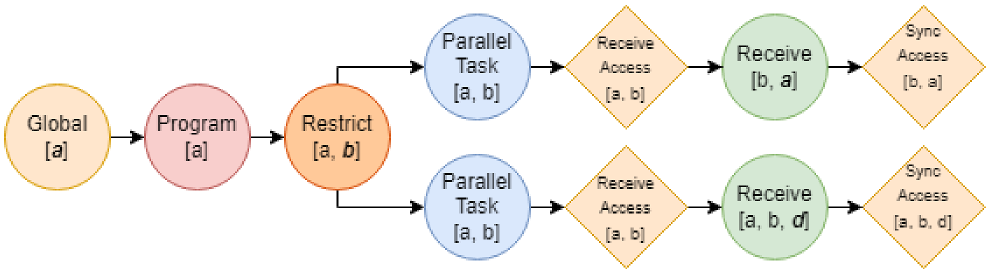


Figure 4.16: Graph of scopes and available names of the program in **Figure 4.15**.

In **Figure 4.16** we can see that the free name *a* defined in the global scope is being shadowed by a new bound name *a* which is received in the top parallel task's receive block. When a name is shadowing a previously defined name, the order in which names appear in the list of available names changes. We can see this as the list no longer begins with *a*, as the shadowing name was the last introduced name, hence *a* is now at the end of the list.

It is important to note that in **Figure 4.15** the first receive block accesses a name *a*. At this point, the name *a* is still a free name, and not the bound name *a* introduced by the receive block.

4.3.2 Scope State Management

One major challenge with the scope manager is the fact that it requires new data to be added as the user defines new names and processes, while making sure we do not accidentally duplicate data. This was especially tricky due to the fact that Blockly's workspace refreshes with any interaction the user makes with the canvas, therefore we cannot simply add new data as it is given by the user.

Due to the nature of the name accessor and process blocks being dynamic based on user defined data, we must be able to refer back to this data during the *block generation* step, which is when Blockly updates the state of the entire canvas, however the data is only collected in the *code generation* step, which follows *block generation*.

The approach used is a dual-state solution where the scope manager maintains two copies of the state, *current* and *previous*. The current state is the fresh state that is being updated as the code generation progresses and the scope manager collects the name, scope and process information. Once the code generation step is finished, and the Blockly canvas is idle until another user interaction occurs, the current state is moved to the previous state and the current state is cleared. This means that once an interaction occurs, the *block generation* step will have access to the previous state, and will be able to maintain valid dropdown names as the dropdown lists will remain properly populated.
One issue with this approach however, is the fact that we always lag 1 cycle behind the changes made by the user, meaning that if the user defined a name, and immediately clicked on a dropdown, more often than not the name would not yet be there. This however, is not a big issue due to the fact that absolutely any interaction with the canvas will update the state appropriately, and the name will then be available.

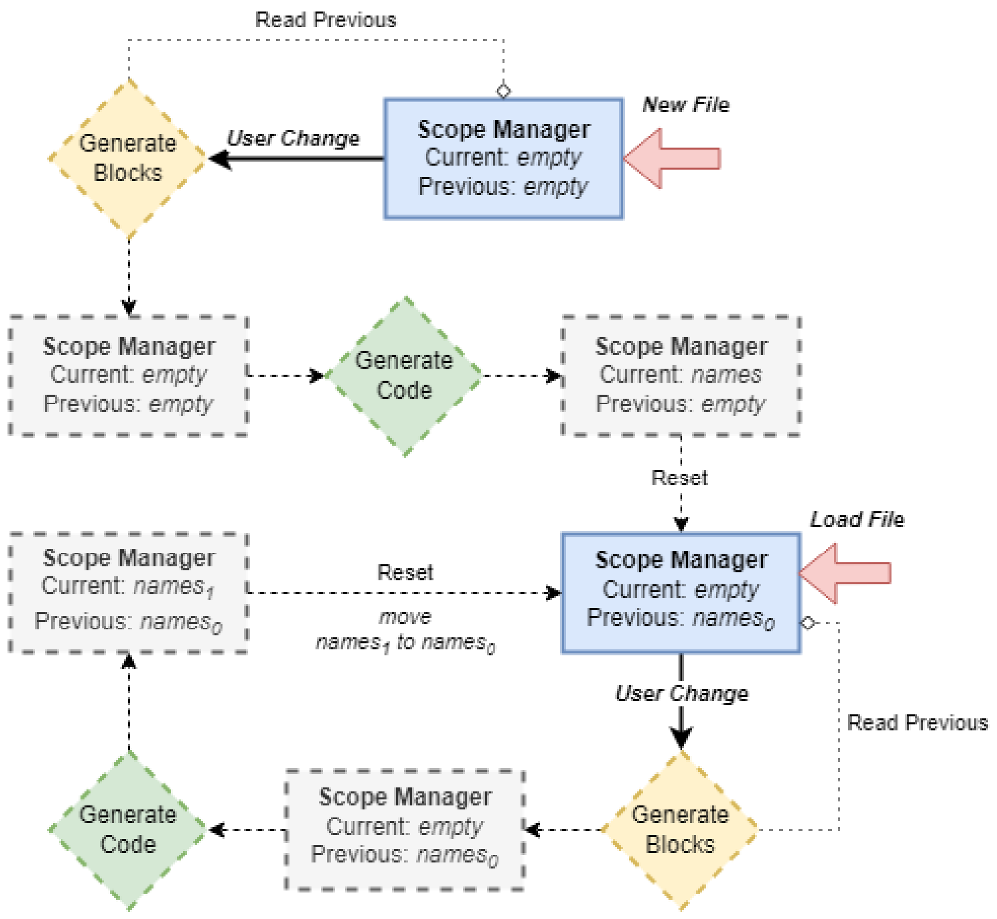


Figure 4.17: State lifecycle of the Scope Manager.

In **Figure 4.17** any states and arrows are either idle states or user-induced actions, and anything that uses a dashed line is a non-interruptible sequence which will go until completion without any way in which the user can prevent it. Dotted lines indicate accessing the data in the scope manager by the block generation step, which requires the previous cycle's data to maintain the correct information in dynamic blocks.

The system has two possible states in which it can initiate, starting with a fresh file, which will perform a single run of block generation and code generation where both states of the scope manager were definitely empty at the beginning. The other option is loading a saved program from file, which will load data directly into the previous state of the scope manager, after which the system proceeds as normal.

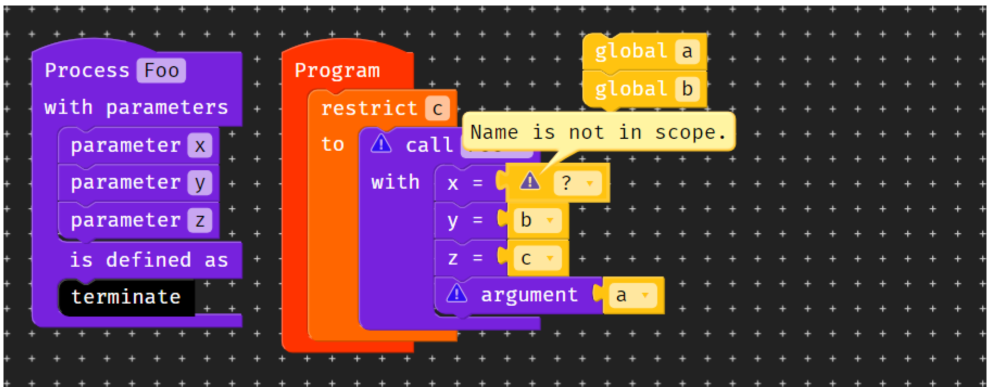
4.4 Syntax Highlighting and Error Management

As mentioned earlier, VisPi enforces certain code style choices in terms of naming of names and processes, namely the required capitalisation of the first character of process identifiers and the lack of capitalisation or an underscore for names. This distinction makes it immediately obvious what each identifier corresponds to, making it possible to perform syntax analysis without the requirement of generating an abstract syntax tree or re-writing a lexer for PIFRA. This provides an opportunity to create a syntax highlighter and a syntax error detection system through a small collection of regular expressions and simple look-aheads or look-behinds.

Block syntax on the other hand, is implicitly controlled by the shape of the blocks and their connections. This leaves only a small subset of blocks which may in fact contain errors, which can be analysed during the block generation step performed by Blockly, as mentioned in the block implementations.

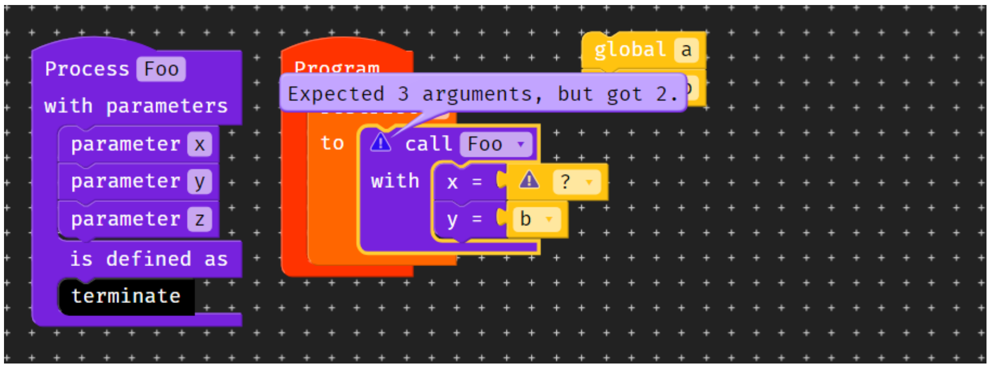
4.4.1 Displaying Warnings and Errors in Blocks

Generally, warnings and errors displayed in the block editor are not critical to the generation of the code output, as any erroneous value can be either omitted or accepted without stopping the code generation process. This decision was made purely on the basis of usability and reducing nuisance as some block editor errors and warnings can be temporary, and making the code generation disappear may be confusing to the end user. Therefore, since the only errors are only semantic, these are displayed to the user as warnings which encourage them to fix any mistakes, while the blocks will still produce the necessary code, even if the names or parameters are invalid.



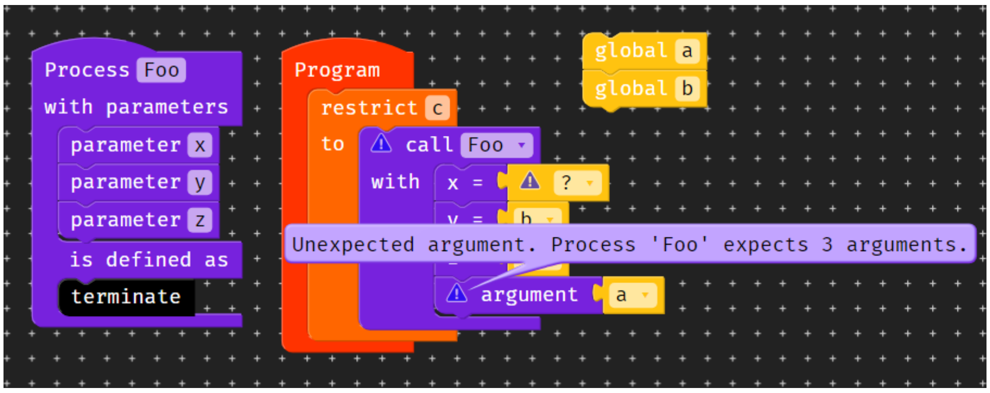
Name is not in scope.

Figure 4.18: Unscoped name warning. This occurs if a name was not chosen (?) or if the name-accessing block was copied and pasted into a scope which does not contain the name, or the name was renamed.



Expected 3 arguments, but got 2.

Figure 4.19: Invalid argument count warning. This is displayed when the number of provided arguments does not match the process definition.



Unexpected argument. Process 'Foo' expects 3 arguments.

Figure 4.20: Unexpected argument warning. This occurs when too many arguments are provided to a process. This warning always appears in conjunction with the warning in Fig. 2.

4.4.2 Regular Expression-based Error Detection

As mentioned previously, due to the enforced style in terms of naming processes and names, it is possible to statically analyse the code output without the need to generate an abstract syntax tree and in fact separate the two systems completely.
Before we explore the regular expressions we will take some time to recap on what is syntactically invalid in PIFRA.

In pi-calculus, and by extension PIFRA, all processes must be terminated explicitly. This means that the last action in any scope or process must be terminal, that being either the termination operator `0` or a call to a defined process or a composition or summation where the substituent parts all terminate at some point. The detection of this requirement is straightforward in that we simply verify using regular expressions that any possible operation other then the aforementioned two, are not final in their scope.

The actions we must consider are the send, receive, restrict and guard. These are all syntax components which require another statement to follow. In fact, all but guard follow the same pattern as all require the sequencing dot operator to follow. Therefore, we can simply define regular expressions and use them with negative-lookaheads to verify that a sequencing operator follows. The regular expressions provided will be given in the form as they are found in JavaScript/TypeScript.

```
[a-z_][a-zA-Z0-9_]*'\<[a-z_][a-zA-Z0-9_]*\>
```

Listing 4.4-1: The Send regular expression.

Please note that the backslash \ is used to escape characters which function as more than simply tokens in the regular expression. The send regular expression is simply the identifier regular expression `[a-z_][a-zA-Z0-9_]*` along with the send syntax elements `'<` and `>`.

```
[a-z_][a-zA-Z0-9_]*\([a-z_][a-zA-Z0-9_]*\)
```

Listing 4.4-2: The Receive regular expression.

Similarly to the prior expression, the *receive* pattern is simply the identifier regular expressions with the parentheses, as dictated by the PIFRA syntax.

```
\$[a-z_][a-zA-Z0-9_]*
```

Listing 4.4-3: The Restrict regular expression.

The *restrict* expression is simply the restriction operator followed by an identifier. It is escaped as the dollar sign symbol is an end-of-sequence marker in regular expressions.

In order to use these expressions for error detection, we must append the negative lookahead verifying that the sequence operator does not follow.

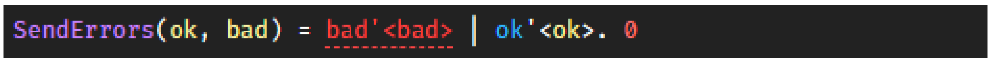
```
(?!\.)
```

Listing 4.4-4: The Negative Look-ahead Sequencing Operator regular expression.

The final error-detection regular expressions are as follows:

```
[a-z_][a-zA-Z0-9_]*'\<[a-z_][a-zA-Z0-9_]*\>(?!\.)
```

Listing 4.4-5: The Send Error regular expression.

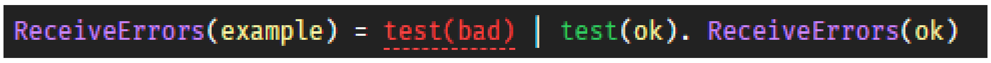


```
SendErrors(ok, bad) = bad'<bad> | ok'<ok>. 0
```

Figure 4.21: Example of an error detected by not following up a send operation.

```
[a-z_][a-zA-Z0-9_]*\([a-z_][a-zA-Z0-9_]*\)(?!\.)
```

Listing 4.4-6: The Receive Error regular expression.

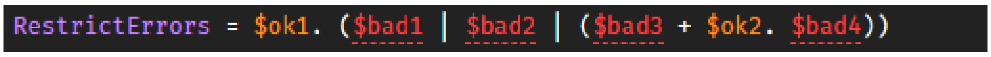


```
ReceiveErrors(example) = test(bad) | test(ok). ReceiveErrors(ok)
```

Figure 4.22: Example of an error detected by not following up a `receive` operation.

```
\$[a-z_][a-zA-Z0-9_]*(?!\.)
```

Listing 4.4-7: The Restrict Error regular expression.

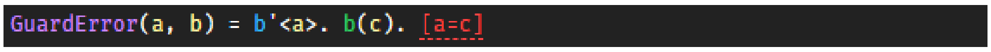


```
RestrictErrors = $ok1. ($bad1 | $bad2 | ($bad3 + $ok2. $bad4))
```

Figure 4.23: Example of errors being detected by `restrict` statements not being followed up by another statement.

```
\[[^\]]+\](?!\s[a-zA-Z_0-9($[])
```

Listing 4.4-8: The Guard Error regular expression.



```
GuardError(a, b) = b'<a>. b(c). [a=c]
```

Figure 4.24: Example of a `guard` statement without a body.

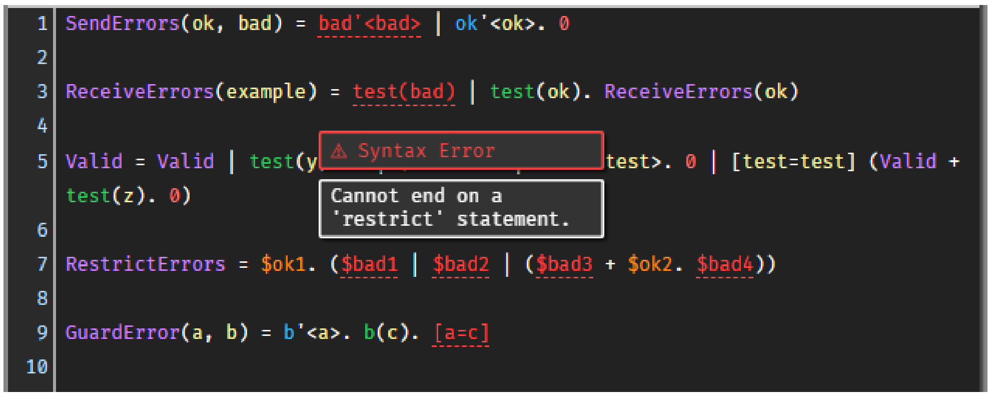
The Guard Error regular expression differs from the prior three due to it not being identifier based nor it requiring the sequence operator to follow. This expression can be split into two parts: the guard clause and the guard body. Recalling the syntax of a guard clause, it can take two forms, the equality variant `[x=y]` or the inequality variant `[a!=b]`. However, for the purposes of the regular expression, and the fact we know that identifiers cannot include brackets, the guard clause can be described as open bracket, at least one character that is not a closing bracket, and the closing bracket.
Now the error we are trying to detect is an empty body of the guard clause. To do this we use a negative lookahead which checks whether the guard clause is not followed by a space and a character which would indicate a body being present. The set of characters $\{a, \dots, z, A, \dots, Z, 0, \dots, 9, \_\}$ checks for a send, receive, process call or termination operations, given their naming requirements. The dollar sign checks for the restrict operation, the parenthesis checks for a parallel or choice composition of multiple processes and the bracket checks for another guard clause.

Finally, an error can be defined as a disjunctive regular expression composition of all errors described above, with the additional use of the named group feature of regular expressions whose use will become apparent in the following section.

```
(?<Error>(SendError|ReceiveError|RestrictError|GuardError))
```

Listing 4.4-9: Combined Error regular expression.

Note: Italicised names used instead of full regular expressions for brevity.



```
SendErrors(ok, bad) = bad'<bad> | ok'<ok>. 0

ReceiveErrors(example) = test(bad) | test(ok). ReceiveErrors(ok)

Valid = Valid | test(y                    test>. 0 | [test=test] (Valid +
test(z). 0)
                      ⚠ Syntax Error
                      Cannot end on a
                      'restrict' statement.

RestrictErrors = $ok1. ($bad1 | $bad2 | ($bad3 + $ok2. $bad4))

GuardError(a, b) = b'<a>. b(c). [a=c]
```

Figure 4.25: Example of hovering over a syntax error in the output window.

4.4.3 Regular Expression-based Syntax Highlighting

Now that we have defined how errors can be detected in the output, we must now detect the correct components of the output. This step, while similar, is more involved than error detection due to the fact we want to highlight individual components rather than whole invalid statements.

```
(?<Symbols>[(),|.+[\]=!'<>\s]+)
```

Listing 4.4-10: The Symbols regular expression.

We want to collect all symbols in the output and apply a consistent colour to them for readability and consistency. The space is included for simplicity.

```
(?<Name>(?<!\$)[a-z_][a-zA-Z0-9_]*(?![('(]))
```

Listing 4.4-11: The Name regular expression.

The name regular expression is used to highlight the usage of names in PIFRA as parameters to processes or arguments to processes, send operations and receive operations. We want to exclusively isolate these instances of names as we otherwise want to use a different highlight. To do this we ensure that the name identifier sequence was not preceded by the restriction operator, and that it is not followed by the send operation's apostrophe or the receive operation's parenthesis.

```
(?<Process>[A-Z][a-zA-Z_0-9]*)
```

Listing 4.4-12: The Process regular expression.

A process is simply an identifier that is capitalised. Since processes cannot be used as arguments, we do not require any further checks.

```
(?<Terminate>\b0)
```

Listing 4.4-13: The Termination regular expression.

The termination or inaction operation is simply defined by the zero character preceded by any word-break, to avoid accidental highlighting of identifiers which include the zero character.

```
(?<Restrict>\$[a-z_][a-zA-Z0-9_]*(?=\.))
```

Listing 4.4-14: The Restrict regular expression.

The restrict regular expression is exactly as described previously. We include the look-ahead for the sequence operator to avoid any possibility of ambiguity with the restrict error regular expression.

```
(?<Send>[a-z_][a-zA-Z0-9_]*(?='))
```

Listing 4.4-15: The Send regular expression

The send regular expression is only targeting the name on which is used as the sending channel, as that is the name which is performing the action, so to speak.

```
(?<Receive>[a-z_][a-zA-Z0-9_]*(?=\()
```

Listing 4.4-16: The Receive regular expression.

Likewise, the receive regular expression is only targeting the name on which another name is being received.

We have now defined all permissible tokens in PIFRA as regular expressions. Again, using the disjunctive composition, we can create a regular expression which will essentially tokenize a line of code into individual tokens.

```
Restrict|Symbols|Terminate|Send|Receive|Name|Process|Error
```

Listing 4.4-17: Tokenization regular expression through disjunctive composition.

Through the use of named groups we can also extract the matched value with the type of token we have matched and apply highlighting accordingly.

4.4.3.1 Tokenization

To tokenize and apply the colours for syntax highlighting we process each line of code individually. Each line is exactly one entire process definition or the system definition.

```
type Token = { type: string; value: string };

const Tokenize = (codeLine: string): Token[] =>
    [...codeLine.matchAll(regex)]
        .map((m) => m.groups)
        .flatMap((g) => Object.entries(g!))
        .filter(([_, value]) => value !== undefined)
        .map(([type, value]) => ({ type, value }));
```

Listing 4.4-18: Tokenization function.

First we match the entire line against our regular expression, and map over all of the detected groups. Since a group is an object of all named groups in our regular expression, calling `Object.entries` returns an array of tuples of key-value pairs, where the key is the group name, or the token type, and the value is the actual string value we will be displaying on the screen. We `flatMap` over those arrays to merge all groups into a single array containing all tokens in order. We then have to filter each token based on whether it contains a value, since each group will still contain all group names but the values for the unmatched groups will be undefined. As the final step we construct each token.

Having the entire list of tokens for a line of code, we can convert each one into HTML and assign the appropriate colour for each token.

```
switch (type) {
    case "Process":
        return "#cc88ff";
    case "Name":
        return "#fff5a9";
    case "Send":
        return "#33bbff";
    case "Receive":
        return "#33cc66";
    case "Restrict":
        return "#ff9512";
    case "Terminate":
        return "#ff7777";
    default:
        return "#ffffff";
}
```

Listing 4.4-19: Syntax colouring function.

4.5 Serialization

Given that VisPi is intended as a tool for learning but also as an alternative workflow tool, the ability to save work and share it is critical. Blockly does support serializing the state of the canvas into a JSON format, however for our purposes we must also store the topology of the program with all the scopes and names within them. Luckily we can simply serialize the state of the Scope Manager along with the canvas state.

```
const Serialize = (workspace: Blockly.WorkspaceSvg, scope:
VispiScopeManager) => {
    const blocks = Blockly.serialization.workspaces.save(workspace);
    const serializationJSON = {
        workspace: blocks,
        state: scope.GetLastScope(),
    };
     return JSON.stringify(serializationJSON);
};
```

Listing 4.5-1: VisPi serialization function, storing both the workspace and the scope information.

When deserializing we simply store the workspace information and state information in local storage under the appropriate keys and reload the window. The default behaviour of the IDE is to load the last state in which the editor was left, meaning that by updating the local storage and reload, we simulate the initial opening of the editor as if the file that was loaded was the last state in which the tool was left in.

```
const Deserialize = (json: string) => {
    const data = JSON.parse(json);
    localStorage.setItem(VISPI_WORKSPACE, JSON.stringify(data.workspace));
    localStorage.setItem(VISPI_STATE, JSON.stringify(data.state));
    window.location.reload();
};
```

Listing 4.5-2: VisPi deserialization function, loading the contents of the file into local storage and reloading the window.

Upon loading the data, important information is extracted and stored within the scope manager which is used for the first instantiation of blocks.

```
const Load = (key: string, scope?: VispiScopeManager): object => {
    const json = localStorage.getItem(key) || "{}";
    if (json.length > 0) {
        ExtractData(json);
    }
    const data = JSON.parse(json);
    const scopeData = localStorage.getItem(key + ":scope");
    if (scopeData && scopeData !== "undefined") {
        scope?.Load(JSON.parse(scopeData));
    }
    return data;
}
```

Listing 4.5-3: The load function. It loads the last saved state from local storage.

```
const ExtractData = (jsonStr: string) => {
    NameAccessStates.length = 1;
    ProcessNames.length = 1;
    const rgx = /"NEW":\s*"(?<NEW>([^"]*))"/g;
    const processRgx = /"PROCESS":\s*"(?<PROCESS>([^"]*))"/g;
    const allMatches = [...jsonStr.matchAll(rgx)];
    const allProcessMatches = [...jsonStr.matchAll(processRgx)];
    for (const match of allMatches) {
        const name = match.groups?.NEW;
        if (name) NameAccessStates.push([name, name]);
    }
    for (const match of allProcessMatches) {
        const process = match.groups?.PROCESS;
        if (process) ProcessNames.push([process, process]);
    }
};
```

Listing 4.5-4: Extraction function, restoring the names and processes for initial block instantiation to prevent data loss.

In **Listing 4.5-4** we see the reference to the `NameAccessStates` and `ProcessNames` variables. These are the global variables used by the name access blocks and process call blocks as seen in **Listing 4.2-7** and **Listing 4.2-56** respectively.

To prevent data loss through accidental closing of the editor or through other measures, the state of the program is saved 500ms after every time there is a change made by the user, this way there is no way the user will lose their work unless they intentionally remove the data. The timeout for saving is debounced such that rapid edits do not cause any noticeable lag, and only the most recent update is saved.

```
const Save = (
     workspace: Blockly.WorkspaceSvg,
     key: string,
     scope?: VispiScopeManager) =>
{
    if (scope) {
        const scopeJson = JSON.stringify(scope?.GetLastScope());
        localStorage.setItem(key + ":scope", scopeJson);
    }
    const ws = Blockly.serialization.workspaces.save(workspace) || {};
    const json = JSON.stringify(ws);
    if (json !== "{}") localStorage.setItem(key, json);
};
```

Listing 4.5-5: VisPi save function which stores the workspace and scope information into local storage.

4.6 Web Application

The web application was implemented using the React Framework [16] with TypeScript. The styling for the application was done using CSS.
The application is a single page website with a static, non-scrollable layout to prevent the elements from moving around on the screen as the user interacts with it.

The application is made up of three main components, the navigation bar, the block editor panel and the output panel.

When designing the application, usability and accessibility were considered, ensuring that the application can support various form factors in terms of screens as well as ensuring that the interactable components on the page are clearly highlighted when navigating with a mouse or keyboard. Additionally, all file options in the navigation bar can also be used through keyboard shortcuts, which can be viewed by hovering over the corresponding button.

4.6.1 Navigation Bar

The navigation bar is the hub for all file related actions in the editor. It offers the user to create new VisPi files and name them, open saved files or save the current program as a file. Additionally, it offers the option to automatically export the current output as a PIFRA file.
The bar also offers some examples of premade VisPi programs and links to useful information such as a How-to Guide for using the tool or the PIFRA GitHub page.

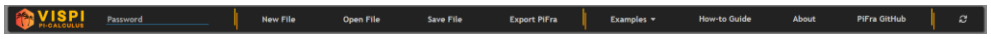


Figure 4.26: VisPi navigation bar.

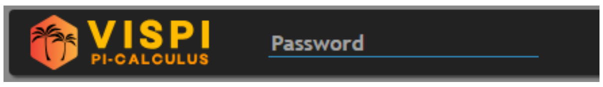


Figure 4.27: Navigation bar file name input box. Current program name is called 'Password'.

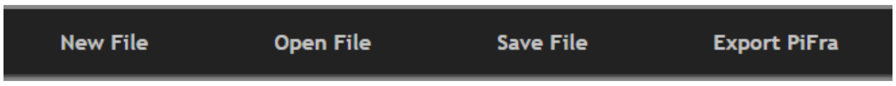


Figure 4.28: Navigation bar file options.

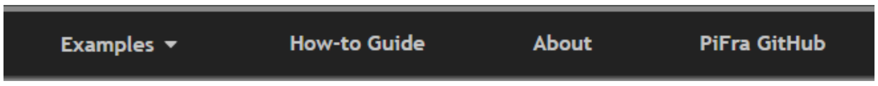


Figure 4.29: Navigation bar help section.

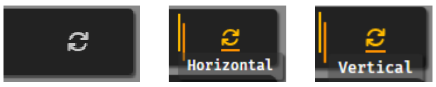


Figure 4.30: Layout button, allows for changing the arrangement of the block editor and the output panel to facilitate both vertical and horizontal screens.

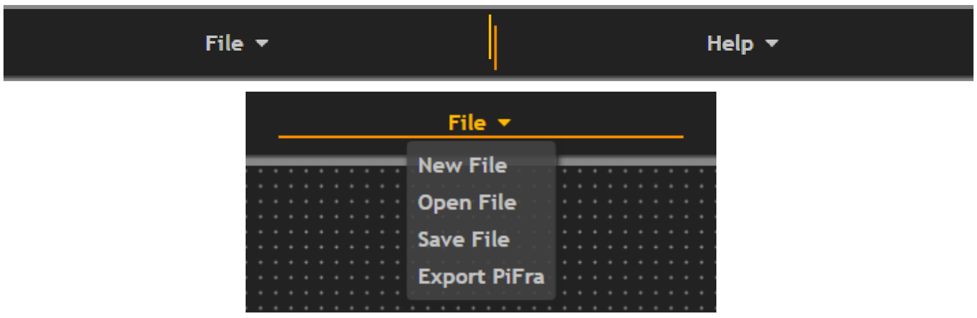
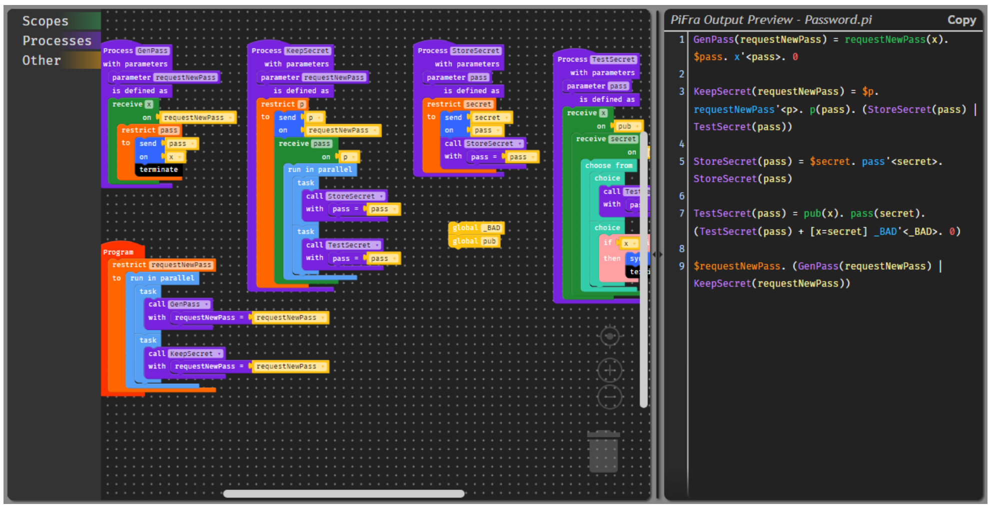


Figure 4.31: File and help sections collapsed to dropdowns on smaller screens.

4.6.2 Editor Layout

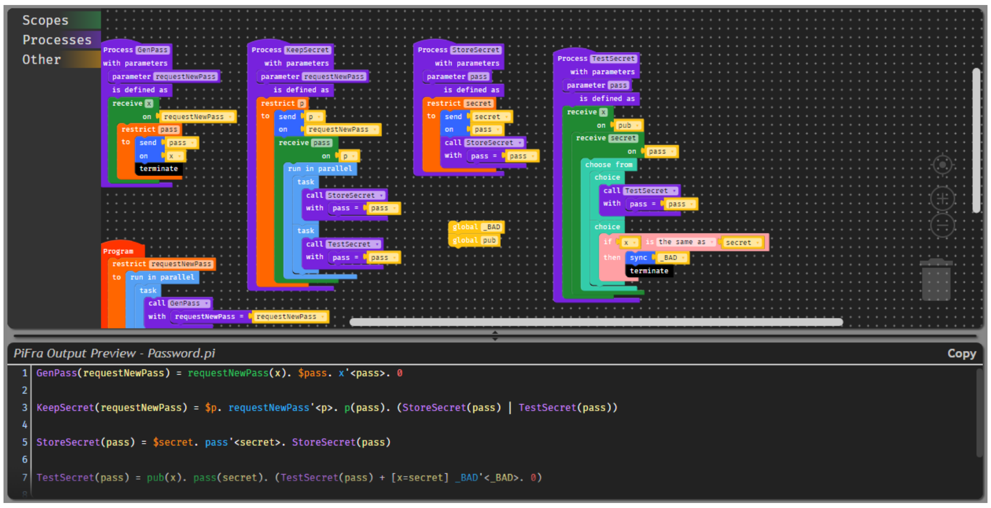
The layout of the editor is simply a two-panel UI as is standard for block-based languages. The editor was made to be flexible to the user's needs so that it can be arranged according to the screen size by rotating the layout from the default horizontal to vertical, and also allowing the user to define how much space they wish to allocate to the block editor and the output panel by using the draggable splitter between the panels.

The size of the monitor matters only in the horizontal axis, as the editor will always take up the entirety of the vertical space it has been given, meaning that vertically aligned monitors can make full use of the extra space in the vertical layout.



```
GenPass(requestNewPass) = requestNewPass(x).
$pass. x'<pass>. 0

KeepSecret(requestNewPass) = $p.
requestNewPass'<p>. p(pass). (StoreSecret(pass) |
TestSecret(pass))

StoreSecret(pass) = $secret. pass'<secret>.
StoreSecret(pass)

TestSecret(pass) = pub(x). pass(secret).
(TestSecret(pass) + [x=secret] _BAD'<_BAD>. 0)

$requestNewPass. (GenPass(requestNewPass) |
KeepSecret(requestNewPass))
```

Figure 4.32: Default horizontal editor layout.



```
GenPass(requestNewPass) = requestNewPass(x). $pass. x'<pass>. 0

KeepSecret(requestNewPass) = $p. requestNewPass'<p>. p(pass). (StoreSecret(pass) | TestSecret(pass))

StoreSecret(pass) = $secret. pass'<secret>. StoreSecret(pass)

TestSecret(pass) = pub(x). pass(secret). (TestSecret(pass) + [x=secret] _BAD'<_BAD>. 0)
```

Figure 4.33: Vertical editor layout.

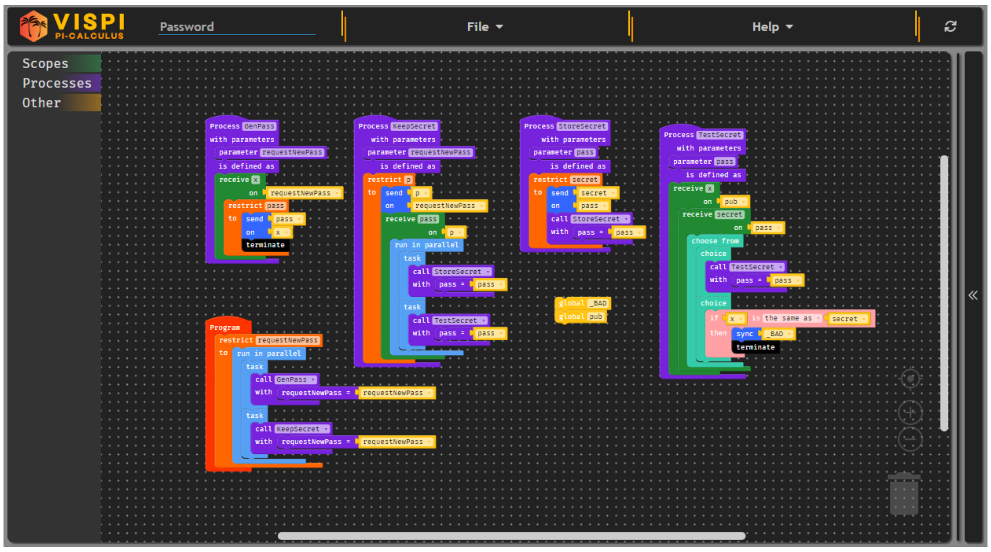


Figure 4.34: Editor with the output panel collapsed providing more space for blocks.

5 Evaluation

This section will cover some of the examples used to verify the correctness of the language, its ability to encode any pi-calculus construct and a brief discussion on the final implementation of the system and language.

The testing of the system was done manually for the most part given the nature of the software and the complexity of setting up examples programmatically for testing the language, as well as time constraints when developing the software.

5.1 Code Generation Correctness

Overall, the language is able to generate code that is capable of representing all possible constructs accepted by PIFRA without any code ever being generated which contains syntactic errors which are not indicated by the IDE. The code generated is always minimised to only include the required number of parentheses and never any excess, even though PIFRA would accept such code.

Example J: Tzevelekos Model

This example covers the *Tzevelekos* model used in *Listing 8.1* in the PIFRA [2] paper, based on the example from FRA paper [15].

```
P(a,b) = a'<b>.$c.P(b,c)
$b.P(a,b)
```

Listing 5.1-1: PIFRA implementation of the Tzevelekos model (ground truth).

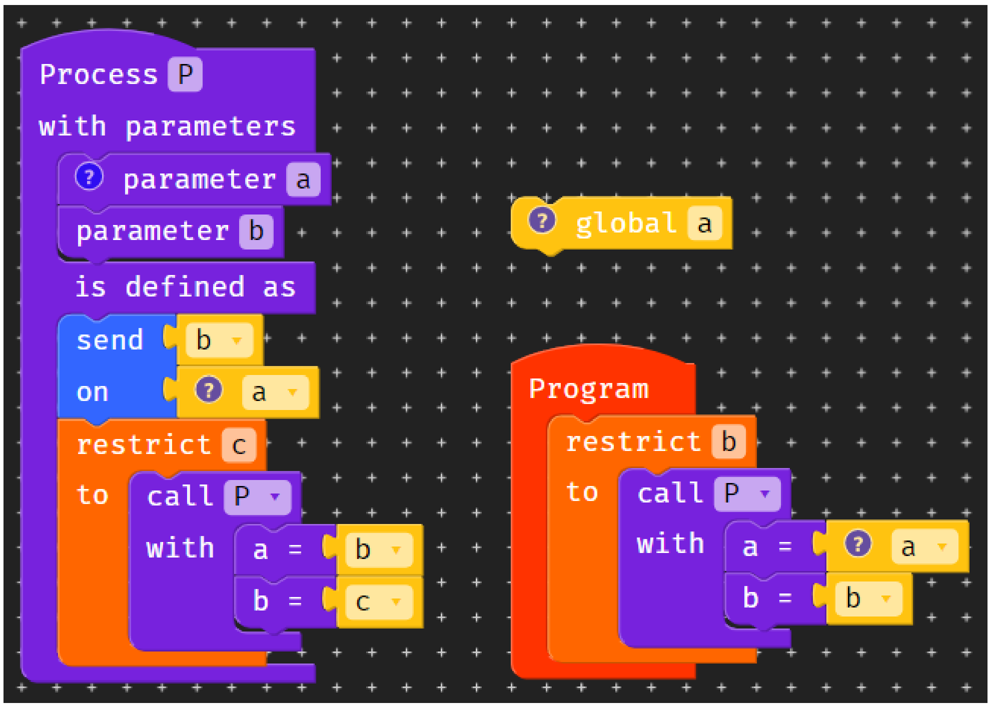


Figure 5.1: VisPi implementation of the Tzevelekos model.

```
P(a, b) = a'<b>. $c. P(b, c)

$b. P(a, b)
```

Listing 5.1-2: PIFRA output from the VisPi implementation in **Figure 5.1**.

```
P(a, b) = a'<b>. $c. P(b, c)
 
$b. P(a, b)
```

Listing 5.1-3: Difference between the ground truth and VisPi output of the Tzevelekos model.

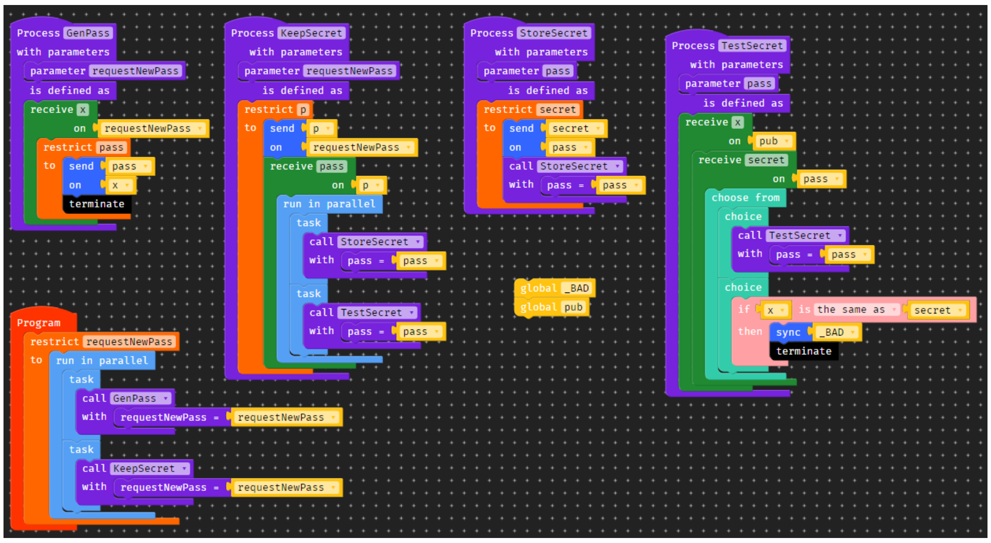
The Tzevelekos model is easily encoded in VisPi and we can see in the output that it matches the ground truth perfectly with the additions of spaces and a new line dividing the process definition and system definition. The differences are indicated by turquoise, which are additions made in the VisPi output.

Example K: Password System

This example covers the *Password* system model presented in the PIFRA paper [2], *Listing 8.16*.

```
GenPass(requestNewPass) = requestNewPass(x). $pass. x'<pass>.0

KeepSecret(requestNewPass) = $p. requestNewPass'<p>. p(pass). (
StoreSecret(pass) | TestSecret(pass) )

StoreSecret(pass) = $secret. pass'<secret>. StoreSecret(pass)

TestSecret(pass) = pub(x). pass(secret). ( TestSecret(pass) + [x=secret]
_BAD'<_BAD>.0 )

$requestNewPass. (GenPass(requestNewPass)  |  KeepSecret(requestNewPass))
```

Listing 5.1-4: Password model from PIFRA (ground truth).



```
Process GenPass
with parameters
  parameter requestNewPass
  is defined as
  receive x
        on requestNewPass
    restrict pass
    to send pass
       on x
       terminate

Process KeepSecret
  with parameters
  parameter requestNewPass
     is defined as
  restrict p
  to send p
     on requestNewPass
     receive pass
              on p
       run in parallel
         task
           call StoreSecret
           with pass = pass
         task
           call TestSecret
           with pass = pass

Process StoreSecret
    with parameters
  parameter pass
      is defined as
  restrict secret
  to send secret
     on pass
     call StoreSecret
     with pass = pass

global _BAD
global pub

Process TestSecret
    with parameters
  parameter pass
      is defined as
  receive x
          on pub
    receive secret
                on pass
      choose from
        choice
          call TestSecret
          with pass = pass
        choice
          if x is the same as secret
          then sync _BAD
               terminate

Program
  restrict requestNewPass
  to run in parallel
       task
         call GenPass
         with requestNewPass = requestNewPass
       task
         call KeepSecret
         with requestNewPass = requestNewPass
```

Figure 5.2: VisPi implementation of **Listing 5.1-4**.

```
GenPass(requestNewPass) = requestNewPass(x). $pass. x'<pass>. 0

KeepSecret(requestNewPass) = $p. requestNewPass'<p>. p(pass).
(StoreSecret(pass) | TestSecret(pass))

StoreSecret(pass) = $secret. pass'<secret>. StoreSecret(pass)

TestSecret(pass) = pub(x). pass(secret). (TestSecret(pass) + [x=secret]
_BAD'<_BAD>. 0)

$requestNewPass. (GenPass(requestNewPass) | KeepSecret(requestNewPass))
```

Listing 5.1-5: PIFRA output from **Figure 5.2**.

```
GenPass(requestNewPass) = requestNewPass(x). $pass. x'<pass>. 0

KeepSecret(requestNewPass) = $p. requestNewPass'<p>. p(pass).
( StoreSecret(pass) | TestSecret(pass) )

StoreSecret(pass) = $secret. pass'<secret>. StoreSecret(pass)

TestSecret(pass) = pub(x). pass(secret). (TestSecret(pass) + [x=secret]
_BAD'<_BAD>. 0 )

$requestNewPass. (GenPass(requestNewPass)  |  KeepSecret(requestNewPass))
```

Listing 5.1-6: Difference between ground truth and output.

By closely inspecting the difference between the ground truth and the output from VisPi, we can see that the output is the same with the only differences being spaces, which were inconsistent in the ground truth. The red highlight indicates a character being in the ground truth, but not in the VisPi output, and vice versa for the turquoise highlights.

Example L: Boolean Binary Buffer

This example will cover the simple Binary Buffer model given as an example in the original Pi-Calculus book [1].

The system is defined as follows as seven process identifiers *Buff* where

$$S = \{ \epsilon, 0, 1, 00, 01, 10, 11 \}$$

$$Processes = \{ Buff_i \mid i \in S \}$$

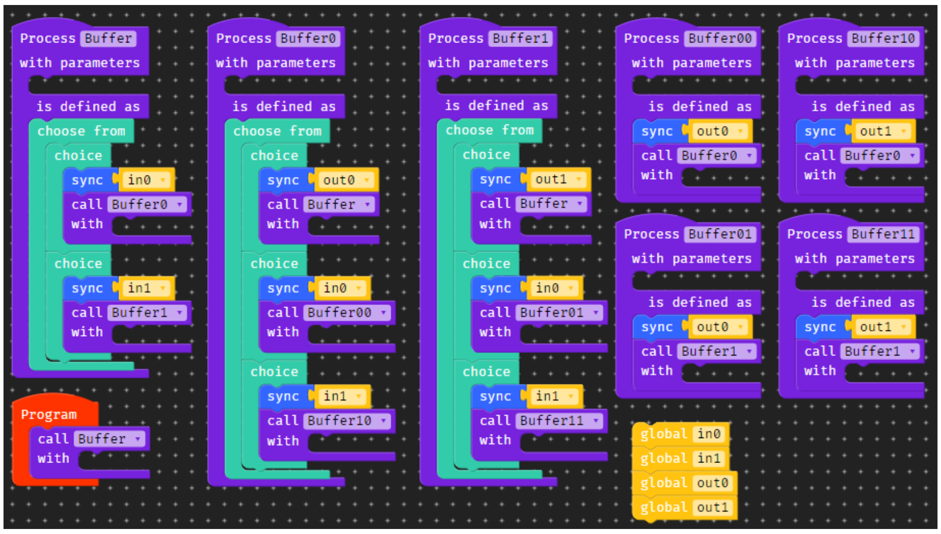
Each value in *s* describes the sequence stored in the buffer, for example $Buff_{01}$ is the buffer where 1 was stored followed by a 0.

$$Buff = \textstyle\sum_{i\in\{0,1\}} in_i.\ Buff_i$$

$$Buff_i = out_i.\ Buff + \textstyle\sum_{j\in\{0,1\}} in_j.\ Buff_{ji}$$

$$Buff_{ij} = out_j.\ Buff_i$$

The summation ($\sum$) follows the summation operator + in pi-calculus.



```
Process Buffer
with parameters
  is defined as
  choose from
    choice
      sync in0
      call Buffer0
      with
    choice
      sync in1
      call Buffer1
      with

Program
  call Buffer
  with

Process Buffer0
with parameters
  is defined as
  choose from
    choice
      sync out0
      call Buffer
      with
    choice
      sync in0
      call Buffer00
      with
    choice
      sync in1
      call Buffer10
      with

Process Buffer1
with parameters
  is defined as
  choose from
    choice
      sync out1
      call Buffer
      with
    choice
      sync in0
      call Buffer01
      with
    choice
      sync in1
      call Buffer11
      with

Process Buffer00
with parameters
  is defined as
  sync out0
  call Buffer0
  with

Process Buffer10
with parameters
  is defined as
  sync out1
  call Buffer0
  with

Process Buffer01
with parameters
  is defined as
  sync out0
  call Buffer1
  with

Process Buffer11
with parameters
  is defined as
  sync out1
  call Buffer1
  with

global in0
global in1
global out0
global out1
```

Figure 5.3: The Boolean Binary Buffer model in VisPi.

```
Buffer = in0'<in0>. Buffer0 + in1'<in1>. Buffer1

Buffer0 = out0'<out0>. Buffer + in0'<in0>. Buffer00 + in1'<in1>. Buffer10

Buffer1 = out1'<out1>. Buffer + in0'<in0>. Buffer01 + in1'<in1>. Buffer11

Buffer00 = out0'<out0>. Buffer0

Buffer10 = out1'<out1>. Buffer0

Buffer01 = out0'<out0>. Buffer1

Buffer11 = out1'<out1>. Buffer1

Buffer
```

Listing 5.1-7: PIFRA output from VisPi of the Boolean Binary Buffer.

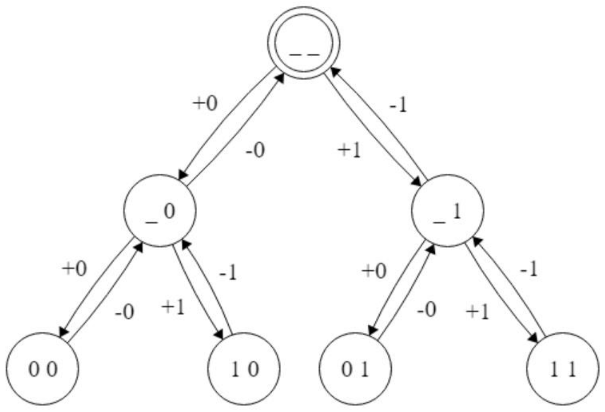


Figure 5.4: LTS output from PIFRA using the output from **Listing 5.1-7**. Relabelled to be more human readable.

6 Conclusion

The main objective of this dissertation was the design and implementation of a visual language capable of transpiling down to PIFRA pi-calculus notation. This objective was fully achieved in combination with the entire implementation of a web application IDE, syntax highlighting for the output code which matches the colour of the corresponding blocks, serialization to files for saving and loading models and a name scoping system ensuring better name safety and program correctness.

6.1 Achievements

With the help of the Blockly library, we were able to design and implement simple blocks which together are capable of expressing any construct in extended pi-calculus. The blocks naturally compose in a way which makes scoping apparent visually, providing visual cues on where names are being restricted.

The code transpilation has been tested thoroughly, and the language only every produces syntactically correct PIFRA code, allowing syntactic errors only through incomplete statements, hopefully improving the learning aspect of using the application by learners easier and more efficient, and also providing the assurance that the output is definitely adhering to the the syntax rules without having to think about making everything is correct or waiting for PIFRA to return an error.

The overall application is relatively simple in the positive sense of the word, as it contains the basic required features necessary for most use cases for writing models with VisPi, saving and loading them, not only allowing for persistence of work, but allowing the block-code to be shared.

The output code, along with blocks themselves, provide enough meaningful error messages to guide learners through the process of creating valid pi-calculus models. Warnings in blocks provide an instant indication that a copy-paste error could've occurred or too many arguments being provided to a process call, while the warnings in the output not only inform the user of incomplete statements, but also inform the user of what the issue is, helping novices understand and learn the syntax of pi-calculus through exploration of the tool.

The name-scope system is a great achievement in the project as it is a major contribution to ensuring the intended behaviour is correctly represented in the models. Not only does the system provide the user only with names which are in-scope, it ensures that typo errors, or unintended declaration of free names by mistake.

The addition of syntax highlighting which matches the colours of blocks which produced each piece of code helps with relating the blocks with the output code, facilitating a transition from blocks to text over time.

The application and language is easily accessible through any browser, making it platform independent and easy for use and collaboration, and hopefully proves itself useful to learners of pi-calculus, researchers preferring to create models more visually and in general help facilitate the teaching of concurrent and parallel process reasoning and design.

6.2 Challenges

One of the main challenges faced in this project was the actual design of each block in an intuitive way. The design of many blocks was an iterative process and each block underwent many changes, often inspired by discussions with Dr. Koutavas. The shapes of the blocks underwent many changes before finally arriving at the main design of blocks which focused on highlighting the scoping properties of certain actions. Another challenge involved deciding whether the blocks should be relatively close to actual pi-calculus or be more higher-level abstractions. Due to time constraints, there wasn't enough time to come up with useful abstractions over common groupings of blocks, however it did lead to block variations and patterns which proved useful in the final implementation of the language.

Many challenges also were encountered due to working with Blockly without prior experience with it. Many features of the library were insufficiently documented or had outdated documentation, requiring alternative approaches to be taken instead.

Designing the scope management system was one of the greatest challenges for multiple reasons. Firstly the process and representation of scopes and ensuring that each scope is uniquely represented with the ability to replicate the scopes, which was required for deserialization, such that the scopes could be restored to their correct state as before the serialization. In addition to that, the major difficulty was implementing the system within the constantly refreshing Blockly environment, which created issues with re-instantiation of blocks dropdowns which had values selected already, but with re-instantiation, the dropdowns wouldn't contain the value required, and would result in an error. Additionally, the refreshing caused duplication of data, which led to the double-state solution.

Along the way, many bugs were encountered with the program, both with the custom systems like the scope manager system, as well as with the Blockly library itself. Problems with custom systems were usually easily resolved due to the clean coding approach and making sure things focused on their given task, making them easily testable. In terms of Blockly bugs, many involved blocks in the canvas moving in undesirable ways when dragging blocks over each other due to the custom detachment functions. These problems were solved through extensive research into the Blockly API to find solutions which help prevent such events from occurring.

Finally, one of the initial challenges was the process of learning pi-calculus concepts in a relatively short amount of time without any prior experience with it or parallel system modelling and analysis. This was a requirement that stalled initial progress on the implementation given that the majority of the system required some understanding of pi-calculus and its constructs and how they interact.

6.3 Future Work

VisPi is currently a standalone tool, however future work could entail integration with PIFRA and other systems to create a unified pi-calculus exploration tool that is suitable for beginners and professionals alike.

The tool could explore the possibility of back-transpilation from PIFRA to VisPi, either as an alternative serialization method and a way to easily convert between the two representations for convenience.

On top of that, the language itself could see additional improvements, such as adding abstractions over current blocks, making the language more approachable to learners already familiar with generic programming languages.

The PIFRA tool could itself see a revision to its graph generation, making the output more user friendly as currently the labels on the graphs are complicated and not approachable to anyone who isn't an expert in the system already.

Finally, given the time limitations of this project being a two semester endeavour, it was expected that the research and study components of the project will not be the primary areas of focus. That being said, a study on the effectiveness of the visual approach to writing pi-calculus programs versus the textual counterpart could provide meaningful insight into what could be improved and whether the visual language provides the necessary aid in learning pi-calculus as opposed to standard textual approaches.

7 Bibliography

[1] R. Milner, Communicating and Mobile Systems: The Pi-Calculus, USA: Cambridge University Press, 1999.

[2] S. Leung, "Modelling Concurrent Systems: Generation of Labelled Transition Systems of Pi-Calculus Models through the Use of Fresh-Register Automata," 2020.

[3] N. Fraser, Q. Neutron, E. Spertus and M. Friedman, "Blockly - Visual Programming Editor," Google, [Online]. Available: https://developers.google.com/blockly.

[4] Scratch Foundation, "Scratch," [Online]. Available: https://scratch.mit.edu/.

[5] B. Broll, A. Lédeczi, P. Volgyesi, J. Salli, M. Maroti, A. Carrillo, S. L. Weeden-Wright, C. Vanags, J. D. Swartz and M. Lu, "A Visual Programming Environment for Learning Distributed Programming," in *SIGCSE '17: Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, 2017.

[6] K. G. Larsen, P. Pettersson and W. Yi, "UPPAAL," Department of Information Technology, Uppsala University, Sweden; Department of Computer Sciene, Aalborg University, Denmark, [Online]. Available: https://uppaal.org/.

[7] J. Mönig and B. Harvey, "Snap! Build Your Own Blocks," University of California, Berkley, [Online]. Available: https://snap.berkeley.edu/.

[8] S. Böhm and M. Běhálek, "Kaira: Modelling and Generation Tool Based on Petri Nets for Parallel Applications," in *UKSim 13th International Conference on Modelling and Simulation*, 2011.

[9] F. Chan, J. Cao, A. T. S. Chan and K. Zhang, "Visual programming support for graph-oriented parallel/distributed processing," 2005.

[10] J. L. Quiroz-Fabián, G. Román-Alonso, M. A. Castro-García, J. Buenabad-Chávez, A. Boukerche and M. Aguilar-Cornejo, "VPPE: A

Novel Visual Parallel Programming Environment," *International Journal of Parallel Programming,* 2019.

[11] E. Pasternak, R. Fenichel and A. N. Marshall, "Tips for Creating a Block Language with Blockly," *2017 IEEE Blocks and Beyond Workshop,* pp. 21-23, 2017.

[12] A. J. Ko, B. Myers and H. Aung, "Six Learning Barriers in End-User Programming Systems," *IEEE Symposium on Visual Languages,* 2004.

[13] M. C. Jadud, "A First Look at Novice Compilation Behaviour Using BlueJ," *Computer Science Education vol. 15, no 1.,* 2005.

[14] R. Milner, A Calculus of Communicating Systems, Berlin, Heidelberg: Springer-Verlag, 1982.

[15] N. Tzevelekos, "Fresh-Register Automata," *SIGPLAN Not. 46.1,* pp. 295-306, 2011.

[16] Meta Open Source, "React - Using TypeScript," 2024. [Online]. Available: https://react.dev/learn/typescript.

8 List of Figures

9 Listings

10 List of Examples

Source Code

The source code for VisPi, additional examples and documentation can be found at `https://github.com/DominikGuzowski/vispi`.