An Introduction to NPL

Suman Karumuri
Hyderabad,India
suman.karumuri@gmail.com

ABSTRACT

Even in this Internet age, reading code is quite hard. Existing Program Visualisation systems could not make visualisation of code in existing languages any easier because the semantics of the languages were not designed with code visualisation in mind.

NPL is an experimental procedural language designed to aid program visualisation and is based on a paradigm called *"Connect Oriented Paradigm"* in which the control and data flow in a program are unified into a *Signal* and processing in the program is unified into a *Connect*.

NPL makes it easy to read and maintain software as it visualises the execution of code with its semantics intact. Owing of its simplicity and generality, future applications of NPL include interactive testing and debugging of applications, domain specific visualisation and also the ability to model a subset of existing languages.

Keywords

Programming Languages, Compilers and Interpreters, Program Visualisation

1. INTRODUCTION

Reading a novel is difficult. But watching a movie based on the novel is easy. Similarly reading code is quite hard [5][3]. Though the compiler understands what the code would do, it would not help the programmer while he is reading it. Reading code would be a lot easier if the compiler could create a movie out of the code, which we can just watch. Such precise visualisation of code at various levels of detail will help solve many software engineering problems [10]. In this paper, we will see how NPL interpreter can create a precise visualisation of its code.

To understand a program you must become both the machine and the program - Alan J. Perlis Epigram

Intuitive Visualisation of code is a tricky business as it involves careful tracking of data flow, control flow and the processing that goes on in the interpreter by instrumenting it(the machine details). The data obtained in-turn should be mapped back to semantics of the language and actual code to produce an intuitive visualisation (the program).

Accurate reconstruction of the semantics from the low-level execution details is very difficult as the compiler has already morphed our code for efficiency to a point from which the high-level semantics cannot be recovered. Moreover, such a technique is highly dependent on the implementation of the interpreter. Every Program Visualisation system faces this problem because the important low-level execution details are left out of language specifications since code visualisation was not one of the design goals of a language.

Hence, a language designed to aid visualisation must specify the low-level execution mechanisms of the code and the higher level semantics of the language in a coherent way. The careful balance of these contradictory requirements will make it possible to intuitively visualise the low-level execution of the programs with the high-level semantics of the language intact, thus making reading code easier.

NPL is an experimental procedural language designed to aid precise and intuitive visualisation of the execution of code with its semantics intact. To the best of my knowledge, NPL is the first language whose code can be accurately visualised by the interpreter. In this paper we shall see how NPL is able to do it:

- In §2 we will support our claim that languages redesigned to aid visualisation would yield better visualisation than better visualisation tools for existing languages.
- The redesigned language is built on *Connect Oriented Paradigm* which acts as an intermediary representation between higher level semantics and low level execution details of code in §4.2
- Next we will discuss the syntax and semantics of the *NPL language* based on "Connect Oriented Paradigm" in §4. Then we prove our claims that we can visualise code with its semantics, by visualising a few examples in *NPL UI*, the code visualiser for NPL, in §5.
- In §5.4, we will discuss how the NPL language design acts as a foundation for other potential avenues of software visualisation like domain specific visualisations and interactive testing and debugging of programs.

Note: The main contribution of this paper is the ability of a language interpreter to be able to visualise the source code with its semantics. The intuitiveness of the visualisation can be improved by better art work and minor visual enhancements. The specific details of the visualisation are discussed only in passing to concentrate on the design issues.

2. LANGUAGES WITH VISUALISATION

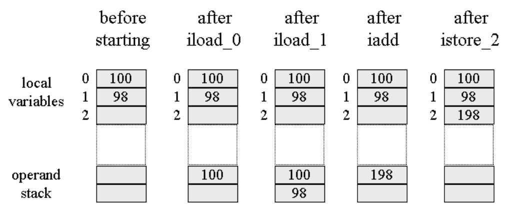
Traditionally, code visualisation tools involve instrumenting the interpreter to gather data about specific aspects of a program. The data thus gathered is presented/visualised to the user in an intuitive UI. Though such tools are good for visualising non-functional aspects of code, they cannot intuitively visualise its functional aspects. The instrumented interpreters usually have a high performance overhead to retrieve everything that is happening in the program and the technique is heavily dependent on the implementation of the interpreter and hence is platform dependent. This dependency leads to inconsistent visualisation of code across platforms and even across various versions of the interpreter, which hinders its adoption. These and various other issues with this design are throughly discussed in [16].

The tools approach is bad because tools like journalists, can report what has happened, but can only guess why it has happened and are usually clueless about how it has happened. For example, Figure 1(a) taken from [18] shows the addition of 100 and 98 in the Java Virtual Machine(JVM). A tool can find out that 100 and 98 were present on the stack, and 198 has been pushed onto the stack finally(the what). But with additional information like the opcode executed, it can guess the operation but can only guess why they are pushed onto the operand stack(the why). But it neither knows how the opcode is executed nor how and why the result has been pushed into the local variable(the how).

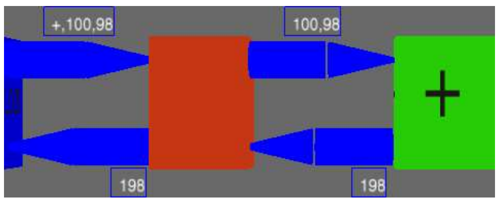
On the other hand the interpreter knows what, why and how something has happened. The same visualisation of addition by the NPL interpreter shown in Figure 1(b) is highly intuitive. Even if you do not know what the red box in the figure does, you can guess from a casual inspection, that it has taken the decision to perform an addition based on the first element of its input. Hence, the interpreters are the ideal candidates to do the visualisation.

But an interpreter is very low-level and cannot show an intuitive visualisation as it does not understand the higher-level semantics of the language. Most of the information useful for an intuitive visualisation has already been discarded during byte-code generation for efficiency reasons. Adding semantic tags to the byte code or designing a new high-level byte code [6] can help aid visualisation to an extent, but those approaches do not work in all scenarios. Code in a language is difficult to visualise because visualisation was not the design goal of the language. The solution to this problem is to re-design the language so that the code written in it is visualisable. But this does not mean the language semantics need to be changed [1]. To facilitate code visualisation the language designer should design a high level interpreter and the semantics of the language based on it. In the words of Martin Rinard [17] he should, "Make systems open while preserving all the good things they do for us".

Designing languages by exposing their execution details is unconventional, as languages are usually designed to hide the execution details from its programmer. But, the stark difference in the quality of visualisation of programs written in a language designed to aid visualisation in Figure 1 is a proof that we can gain a lot without giving too much away. The NPL language discussed next is one such attempt.



(a) Addition in Java VM



(b) Addition in NPL

Figure 1: The figures (a) and (b) show the addition of 2 numbers 100 and 98 in the JVM and the NPL interpreter respectively. Lots of things like the operation being performed and the way it is performed cannot be shown by (a). But in (b) the whole process is intuitive and transparent.

3. NPL

The name *NPL*, a recursive acronym, stands for ***N****PL* ***P****rogramming* ***L****anguage.*

NPL System refers to the NPL Language and it's UI, shown in Figure 2. The NPL Language in-turn has a lexer, a parser which generates byte-code and an interpreter which consumes the byte-code. The UI has an event handler and a visualiser, written using the game engine Panda3D, which visualises code in 3D. The NPL Language and it's UI are designed to complement each other.

The design of NPL (short form for the NPL Language) is discussed in the next section.

Note: All the screenshots are either zoomed in or zoomed out versions of visualisation either to highlight areas of interest or to overcome the limitations of the paper.

4. THE NPL LANGUAGE

NPL is a turing complete, stackless, strongly typed imperative procedural language[1] with run-time type checking, first-class functions and built-in associative arrays. NPL has a prefix syntax like lisp and has the semantics of python (without OO features) like types being attached to data rather than variables. NPL is based on a paradigm called the "Connect Oriented Paradigm".

In this section, we briefly introduce the syntax of NPL with a few examples. The "Connect Oriented Paradigm" which forms the foundation for NPL is discussed next. In the rest of this section following it, we will explain how the

[1]Only the features that can be visualised by NPL UI, as of this writing, are discussed in this paper

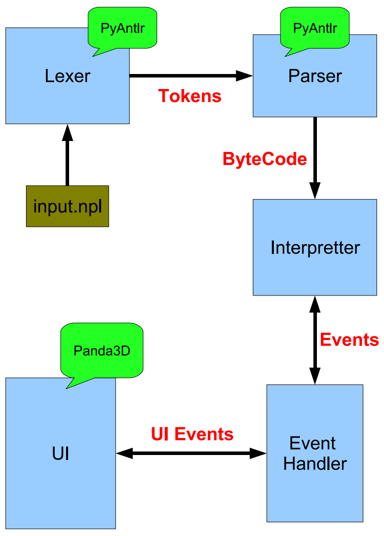


Figure 2: Overview of the NPL system which is implemented in python. The green call-outs show the libraries used for those modules.

low-level execution details and higher level semantics of NPL are implemented based on this paradigm.

Before we look at the syntax of NPL, here are some programs in NPL:

4.0.1 Hello World

```
(
  (print 'Hello World')
)
```

4.0.2 Fibonacci Sequence

```
(
  (= a 0)
  (= b 1)
  (print a)
  (print b)
  (= lim 5)
  (while  (!= lim 0)
          (
              (= c (+ (+ a 0) b))
              (print c)
              (= lim (- lim 1))
              (= a b)
              (= b c)
          )
  )
)
```

4.0.3 Factorial of 3

```
(
 (def fact n
      (
        (if (<= n 0)
            (return 1)
            (return (* n (fact (- n 1))))
        )
      )
  )
  (print (fact 3))
)
```

4.1 Syntax

NPL uses a prefix notation as it's syntax similar to lisp. Prefix notation was chosen as it closely resembles the semantics of the language. Below is the syntax of NPL in EBNF notation.

```
block :  '(' (block)* ')'
|  '(' connect_name (atom)* ')'
|  '(' 'def' (arg)+ block ')'

connect_name : VAR                // Connect name
| 'print' | 'return' | 'while' | 'if' //keywords
| '=' | '+' | '-' | '*' | '/'  //operators
| '>' | '>=' | '<' | '<=' | '!='

atom : NUMBER | STRING | VAR | block

arg : VAR

NUMBER  : //integers and reals
STRING  : '<text>' //quoted strings
VAR     : <text>   // unquoted strings
```

block corresponds to a lisp S-expression and *connect_name* corresponds to a function name in lisp. A lisp programmer would feel at home with the syntax of NPL and an average programmer can easily decipher the meaning of the programs. It should be quite obvious by now that we do not need any special understanding of the underlying paradigm to write programs in NPL. An understanding of the Connect Oriented Paradigm, described in the next section, is only required if you want to understand and interact with the visualisation.

4.2 Connect Oriented Paradigm

"Connect Oriented Paradigm"(COP) was designed as an intermediate representation of programs, such that it is rich enough to explain the semantics of the language and at the same time is low-level enough for the interpreter to execute, without hiding too many execution details. It is a paradigm rather than a model/framework, as it influences the way we think about NPL programs.

In COP, there are *connect*s which communicate by sending *signal*s to each other. They are detailed below:

Signal:

Each *Signal* is an uniquely identified structure in the interpreter which carries the data between Connects. The data on the signal and the destination of the signal determine the data flow and the control flow of the program respectively. Thus, a *Signal* unifies the control and data flow in a program. When a Signal runs, it transfers it's data to the receiver [2] and runs the destination Connect(receiver). A signal terminates itself, after it has run.

[2]If there is no receiver, it creates a default receiver(a GenericConnect)

Connect:

Each *Connect* is a loosely coupled, uniquely identified structure in the interpreter which takes a signal as an input and gives out a signal as output.

During its execution, a connect can create other connects. The type of signal emitted by the connect during its execution depends on the following conditions:

- The type of connect(type not to be confused with type systems in Programming Language Theory)
- The data it has received from the input signal[3]
- The current state of the connect (the connect behaves like a Finite State Machine)

When a connect reaches it's final state it sends a signal to it's sender and terminates itself , after making sure that the connects created by it have also terminated. A connect unifies all the processing that occurs in the interpreter.

All Connects and Signals manage their own memory. COP is better suited for visualisation because, it unifies the control and data flow in a program into a signal which otherwise would be scattered across a program counter, data stack, heap and function call stacks. A Connect abstracts away all the processing and side-effects into a single representation, which otherwise would be embedded in the interpreter logic and byte code logic.

After COP we designed, I found that it is very similar to the Actor model [9]. The Connect and Signal are analogous to an Actor and message respectively. But COP is much low-level than Actor model as Connect and Signal are the atomic blocks of the interpreter where as an Actor and Message are semantic high level objects which are executed under the veil of the interpreter. Next, we will see how the NPL Interpreter implements the low-level details of the interpreter using COP.

4.3 NPL Interpreter

The NPL interpreter executes the byte code generated by the parser. To simplify visualisation, all the low-level functionality of the interpreter has been squeezed into the *Signal* and 3 special Connects: *BlockConnect*, *GenericConnect* and *SymbTableConnect*. Everything else is implemented as a built-in connect or a function inside the interpreter (discussed in the next section). This design makes NPL's Interpreter a distributed interpreter [11] like the ACT 1 interpreter.

Thorough understanding of these 3 connects and the Signal are very important as they define the low-level execution details and some language semantics of NPL. Though these are the implementation details of the interpreter, they provide a concrete representation of the execution of the program, which mirror the image the user has of the execution of the program. Understanding them is necessary to understand the visualisation.

In the following subsections we will see a programmer's perspective (an overview) followed by the technical details of the Signal and the 3 connects in NPL.

4.3.1 Signal

As defined in §4.2 a signal in NPL carries data between connects. If the signal has a pre-determined destination[4], then all the data on the signal will be the data for the destination connect as shown in Figure 3(c).

If the destination of the signal is unspecified by the sender, then the signal creates a GenericConnect§4.3.4 which will decide the destination based on the first data member on the signal as illustrated in Figure 3(a) and Figure 3(b).

4.3.2 BlockConnect

A block in npl, analogous to S-expression in Lisp, is anything between "(" and ")". A *BlockConnect* represents one such block. Blocks in NPL can be nested, so can the BlockConnects. Since blocks can be nested, a BlockConnect[5] can represent the whole program, a single line or a set of lines, an expression or a part of it.

A BlockConnect plays many roles in the interpreter. These roles can be categorised based on its contents into 3 types: nested, unnested and a function. Irrespective of the category, they are visualised in the UI as a blue box with the words "BLOCK" on it by default as shown in Figure 3(a).

A nested BlockConnect is one that contains other BlockConnects as shown in Figure 4(a). A nested BlockConnect represents a set of expressions. It executes each of the BlockConnects in a serial fashion and terminates once all of the nested BlockConnects are executed. When a nested BlockConnect which represents the whole program is run, it serialises the execution of whole program by executing its nested blocks serially.

An unnested BlockConnect is one which does not have a BlockConnect as its first element. For example, Figure 4(b) shows 2 unnested BlockConnects which have 'print' and '+' as their first elements both of which are built-in Connects. The BlockConnects in Figure 4(b) represent the line (print 'Hello World') and the expression (+ 100 b) respectively.

A function in NPL is a nested BlockConnect with a header as shown in Figure 4(c). The header is used to store the function's name and its parameter lists. When a BlockConnect with an optional header is run with out any input arguments it stores the function definition in the Symbol Table. If the same BlockConnect is called with an argument list then it creates an internal symbol table in the function scope, saves argument list and executes the function(runs the nested BlockConnects). The appearance of a function BlockConnect can be customised by a special configuration file (see §5.4.1).

4.3.3 SymbTableConnect

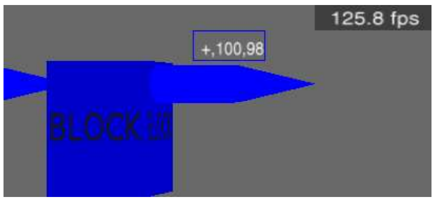
A symbol table in NPL is best described as a dictionary with some additional lookup rules to implement lexical scoping. A *SymbTableConnect* implements a symbol table local to a BlockConnect in NPL.

The SymbTableConnect just implements the rules for lexical scoping and delegates the actual responsibility of storing the values to a DictConnect, the connect which implements a Dictionary. Whenever, a request for a accessing or storing a variable arrives at a SymbTableConnect, it passes on that request to the DictConnect (it's internal dictionary), whose response is sent to the sender. If the DictConnect does not

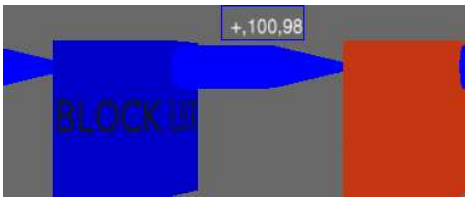
[3]In the present design only the first element of the data is considered

[4]the destination is determined by the sender(connect which gave out the signal)

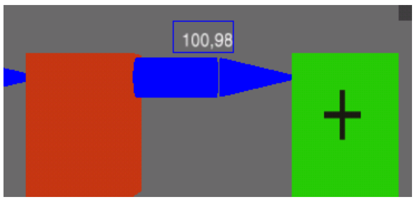
[5]The words BlockConnect and block are synonyms from now on



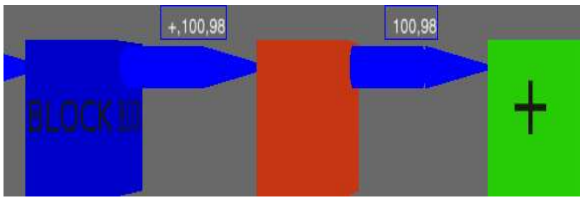
(a) Signal emitted without a destination



(b) After Signal in (a) created a GenericConnect

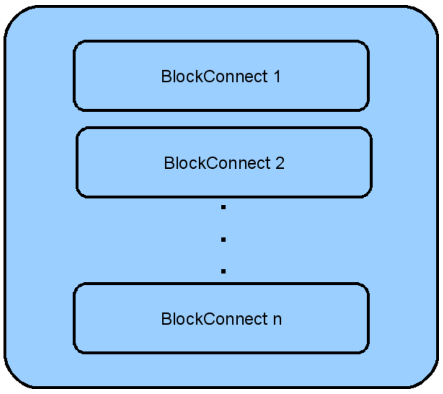


(c) Signal with a predetermined(by GenericConnect) destination

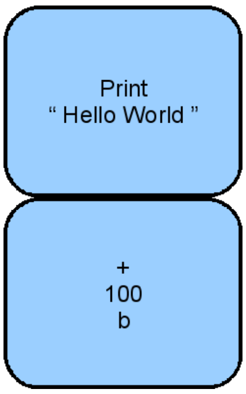


(d) All 3 images

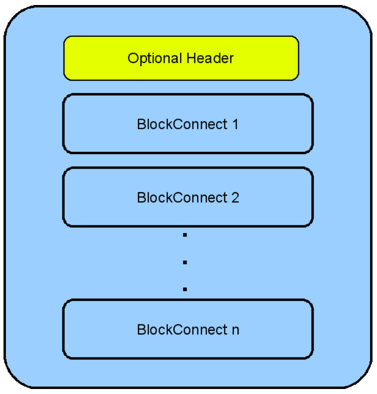
Figure 3: The above images are taken from Figure 1(b). The blue box with text "BLOCK" represents a BlockConnect. The red box represents a GenericConnect. The blue arrow represents a signal with its data on top and pointing towards its receiver.



(a) A nested BlockConnect



(b) 2 Unnested BlockConnects



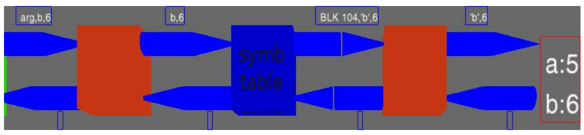
(c) BlockConnect that implements a function

Figure 4: The above images show various ways in which BlockConnect executes.

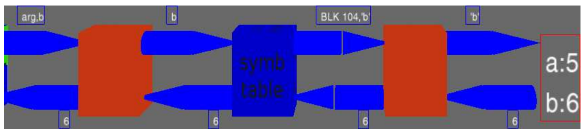
have the value, happens when a global variable is accessed within a function, the SymbTableConnect looks up for the value in it's parent scope.

Figure 5(a) shows storing a new variable 'b' in the symbol table and Figure 5(b) shows accessing the value of variable 'b' from the symbol table. The symbol table in Figure 5 already has a variable 'a' in it.

This design was chosen for implementing the symbol table in the interpreter for 3 reasons. Firstly, to make the semantics of NPL similar to the semantics of python. Secondly this eases adding reflection and meta-programming facilities to NPL in future. Thirdly, no special visualisation is required for symbol table as it can be visualised just like any other dictionary in the UI.



(a) Assignment of value 6 to b



(b) Getting the value of b from the symbol table

Figure 5: The above images show the assignment and retrieval of values from the SymbolTable.

4.3.4 *GenericConnect*

For understanding the visualisation, you can safely assume that a *GenericConnect* does everything in the interpreter that the Signal, BlockConnect and SymbTableConnect do not. It is visualised in the NPL UI as a red box as shown in Figure 3.

The GenericConnect functionally performs the following tasks. Firstly, it looks at the data from the input Signal and decides the output signal depending on the first element on the incoming signal. If the first element on the incoming signal is a function[6], it creates a BlockConnect representing the function body and passes the parameters as values to the BlockConnect. If the first element is a BlockConnect, it sends all the input as data on the output signal to the BlockConnect.

A GenericConnect also manages the creation and destruction of function scopes. The function call mechanism of NPL like python is call-by-value, which is enforced by GenericConnect. Only an overview of the functionality is provided to avoid the unnecessary implementation details.

A Signal with no destination creates a GenericConnect as its destination. They work in unison to provide the required functionality of mapping the name of a function to its implementation. For example, if the data on the signal is (+, 100, 98) with no receiver (no Connect on the destination), the signal creates a GenericConnect which will become its destination. The GenericConnect will look at +, the first element of the data on the signal, finds out that addition should be performed and creates an AddConnect and sends a signal with the rest of the data(100, 98) to the newly created AddConnect. This is illustrated in Figure 3(d).

[6]The function name look up happens from the symbol table. Refer next section for details on built-in connects.

4.4 Interpreter's view of a program

An interpreter looks at an NPL program as a BlockConnect. This BlockConnect contains nested BlockConnects which in turn may be a nested BlockConnect or an unnested BlockConnect (function / built-in connect with some parameters). For example, the "Hello World" program §4.0.1 will be seen by it as shown in Figure 7.

Bootstrapping: The process of running the BlockConnect which represents the whole program, to start the execution of the program, is called Bootstrapping. During this process the interpreter creates a BlockConnect which represents the whole program and a signal to run this BlockConnect. When the interpreter starts running after it is bootstrapped, it runs this signal which runs the BlockConnect. The visualisation shows everything in the UI after the first signal has been executed as in Figure 9(a).

This is all there is to know about the internals of NPL. In the next section we will see some built-in connects, which provide the conventional facilities of an interpreter and will help us write useful programs in NPL.

4.5 Rest of the Language Features

NPL provides built-in connects for Relational and Binary operators, Conditional flow and iteration, I/O facilities, dictionaries etc, the tasks which the NPL interpreter itself cannot do. Figure 6 documents the built-in connects and their functions in NPL. Each of these connects take their data(if any) on the input signal, perform the operation and return the result on the output signal. These built-in connects when used in conjunction with the Signal and 3 Connects discussed in the previous section make a turing complete interpreter that can be used for running useful programs.

All these connects obtain the value of the variables in their input by performing a lookup from the symbol table.

Design Tradeoff: Ideally, all the built-in connects can be implemented as functions in NPL, whose values will be looked up in the symbol table every time a function is invoked. But this lookup for built-in connects complicates the visualisation as the user expects them to be the part of the runtime. To facilitate visualisation, the GenericConnect was designed to decide the Connect to execute for built-in connects, instead of doing a symbol table lookup. Since + is a built-in connect, the GenericConnect has created an AddConnect in Figure 3(d), instead of doing a symbol table lookup to know what + stands for.

Next we will see how programs written using these connects can be visualised in the NPL UI.

5. NPL UI

The *NPL UI* is an interactive direct manipulation User Interface in which an NPL program is visualised in 3D[7].

[7]The visualisation is 2D at present

	Connect Type(Usage)	*Description*
Binary Operators These connects take their operands from input signal, perform an operation and return the result in the output signal .	AddConnect(+)	This connect adds 2 numbers.
	SubtractConnect(-)	This connect subtracts 2 numbers.
	DivConnect(/)	This connect multiples 2 numbers
	MultConnect(*)	This connect divides 2 numbers.
Relational Operators These connects take the operands from the input signal and return a boolean value in the output signal.	GtConnect(>)	This connect performs the greater than comparision of the inputs
	GeConnect(>=)	This connect performs the greater than or equal to comparision of the inputs
	Lt Connect(<)	This connect performs the less than comparision of the inputs
	LeConnect(<=)	This connect performs the less than or equal to comparision of the inputs
	Ne Connect(!=)	This connect performs the checks for the quality of the inputs
Conditional flow These Connects implement the conditional statements.	If Connect(if)	This connect implements the "if "conditional and "if-then-else" conditional. It takes a conditional block as the first parameter. The next block will be executed when the condition is true. The optional 3rd block will be executed when the condition is false.
	While Connect(while)	This connect implements a while loop.The first block is the condition to be executed.The second is the body of the while loop.
I/O Operations These connects perform the I/O operations.	Print Connect(print)	This connect prints the input from the signal.
	Read Connect(read)	This connect reads the input from the console and returns the data read as a signal.
Utility Connects These are some other utility connects that are available.	Equal Connect(=)	This connect performs an assignment operation.It takes a variable and it's value and saves them to the symbol table.
	ReturnConnect(return)	This connect takes care of returning values from a function.
Structres NPL provides associative arrays which can be used as structures and lists.	DictConnect(dict)	This Connect implements an associative array aka dictionary.

Figure 6: Table showing all the built-in connects in NPL

The UI runs off the events generated by the connects and signals running in the interpreter (see Figure 2). Once the events are gathered, the UI visualises the connects and signals in a 3D environment using a layout algorithm. The layout algorithm used currently is a tree layout algorithm with some tree re-writing rules which change the structure and appearance of the tree. The exact details of the algorithm are out of scope of this paper.

In this section, i will give a brief overview on the events that the UI receives after which we will see the visualisation of a few programs in NPL.

5.1 UI Events

The UI runs off the events generated by the NPL interpreter. The information present on the events is nothing but the internal state of the Connects and the Signals. Some example events are shown below. Some of the details on the events are self-explanatory and are not discussed here further as they are gory implementation details.

```
CycleStart

SigCreate:{'sender': '106','name': '110','id': '110',
'scope': 0, 'payload': ['print', {'type': 'STRING',
'value': 'Hello World'}], 'reciever': -9999}

SigRun:{'id': '109'}

ConnCreate:{'sender': '111','last_scope': 0,
'current_state': 0, 'name': '112', 'scope': 0,
'type':'PrintConnect', 'id': '112', 'data_stack':[]}
```

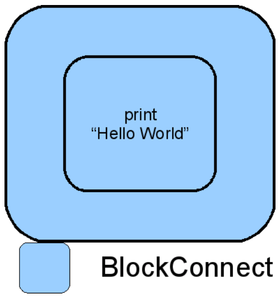


Figure 7: BlockConnect view of the "Hello World" Program

```
ConnRun:{'sender': '108', 'signal': '110',
'current_state': 0, 'data_stack': [],
'id': '106', 'name': '106'}

ConnTerm:{'id': '112', 'name': '112'}

CycleEnd
```

Though this approach is very much similar to the usual approach of instrumenting the interpreter, it differs from the traditional approach as the events contain all the high-level and low-level information there is about the execution of a program. It should also be noted that most of this information is really closer to what the NPL programmer has in his mind regarding the execution of the program rather than some low-level byte-code instructions for a stack based VM.

5.2 Examples

In this section, we will see the visualisations of some programs written in NPL and show how the visualisation depicts the execution and semantics of the program. We have chosen small programs for visualisations of code considering the limitations of this medium.

5.2.1 *Hello World*

The 'Hello World' program in NPL is given below and it's visualisation can be seen in Figure 8. We will see how we got the visualisation by going through the program step by step and finally see if the visualisation matches the view we had in our mind about the execution of the program.

```
(
  (print 'Hello World')
)
```

The parser generates a BlockConnect view of this program (in the byte-code) which looks like Figure 7. This structure is very much similar to our program.

For the sake of explanation, we will use the internal id's of Signals and Connects to refer to a particular object in the UI. This is necessary when we are talking about many Signals and Connects in the same sentence. **These internal id's are not required while viewing the visualisation in the UI, as it is animated. The first element of the signal is the internal id. The rest is the data on the signal. The id for Connects are shown on top of them.**

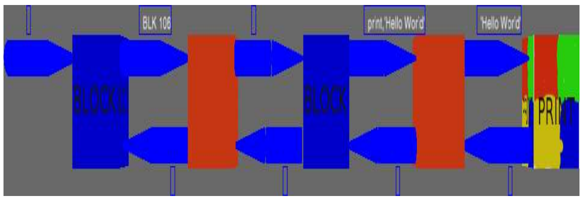


Figure 8: Visualisation of the 'Hello World' program in the NPL UI

- After the interpreter bootstraps, we will see a BlockConnect (102) which represents the whole program and a signal (105). This signal when run executes its destination the BlockConnect (102) .

- The signal (105) runs, executes the BlockConnect (102), it's destination and terminates[8]. Since BlockConnect (102) is a nested BlockConnect, it creates a new BlockConnect[9] (106), gives out a Signal (107) without destination to execute the newly created BlockConnect (106). This is shown in Figure 9(a).

- The signal (107) runs[10] and creates a GenericConnect (108), since it does not have a destination, and runs the GenericConnect. The GenericConnect (108) identifies from the data that the BlockConnect (106) should be run, and gives out a signal (109) for that BlockConnect to execute as shown in Figure 9(b).

- When the Signal(109) is run, it executes the BlockConnect (106), its destination. The BlockConnect (106) runs and gives out a Signal (110) with its contents (print, 'Hello World') as its data, since it is a unnested BlockConnect as shown in Figure 9(c)

- Since the signal (110) has no destination, it creates a GenericConnect (111), transfers the data to it, runs the GenericConnect (111) and terminates. Based on the first element of the data (print), the GenericConnect (111) identifies that data should be sent to a built-in Connect, the PrintConnect. So, it creates a PrintConnect (112) and sends it a signal (113) with its data of other than first element ('Hello World'). This is shown in Figure 9(d).

- When the signal (113) runs, it transfers its data to the PrintConnect and runs the PrintConnect. The PrintConnect prints the data ('Hello World') to the terminal and returns a signal (114) to its sender and terminates as shown in Figure 9(e).

- The signal (114) runs and executes the GenericConnect (111). The GenericConnect (111) intern has nothing to do, so it sends a signal (115) to its sender[11] and terminates. This is shown in Figure 9(f).

[8]The UI is yet to be enhanced to visualise Active and Terminated objects in different ways

[9]The newly created BlockConnects are not shown in the UI until they are needed

[10]All Signals terminate after they are run

[11]Once a Connect has reached its final state it terminates after sending a signal to its sender

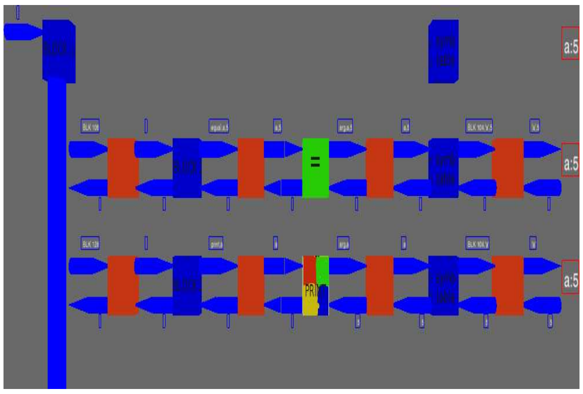


Figure 10: Visualisation of the 'Variable Assignment' program in the NPL UI

- The Signal (115) runs and executes the BlockConnect (106). The BlockConnect (106) runs, sees that it is in its final state since it is an unnested BlockConnect and terminates after sending a signal (116) to its sender. This is shown in Figure 9(g).

- The signal (116) runs and executes the GenericConnect (108). The GenericConnect (108) runs, sees that it is in its final state, sends a signal (117) to its sender and terminates as shown in Figure 9(h).

- The signal (117) runs and executes the BlockConnect (102). This BlockConnect (102) checks to see if there are any more BlockConnects that should be executed since this is a nested BlockConnect. Since this is one line program it has no more nested BlockConnects to execute. Unlike other BlockConnect's this BlockConnect (102) terminates the program by terminating itself, since this BlockConnect represents the whole program.

Now, if you compare the Figure 7 and Figure 9(h), you can see how similar they are. Everything that we know the interpreter does is inside the red boxes and the data and control flow we assume is nicely packed into the blue Signals in Figure 9(h). This figure shows the higher level semantics of the language which are blocks and the lower level execution details of code in a nice way.

You should have already observed that the explanation of the processing in the interpreter was a bit too repetitive. The reason for repetition is that the simple rules of signals and connects involved were used again and again in various contexts, during the execution of the program.

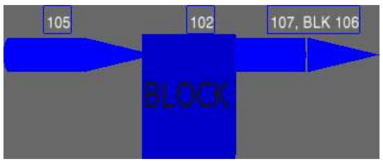
5.2.2 *Variable Assignment Program*

In this section, we will visualise the program shown below.

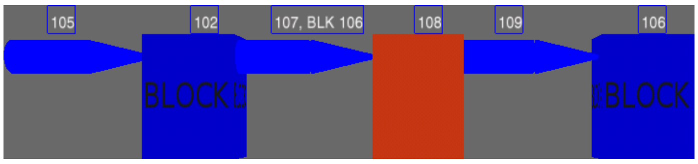
```
(
  (= a 5)
  (print a)
)
```

This program assigns a value 5 to the variable 'a' and then prints the value of 'a'. The complete visualisation of this program can be seen in Figure 10.

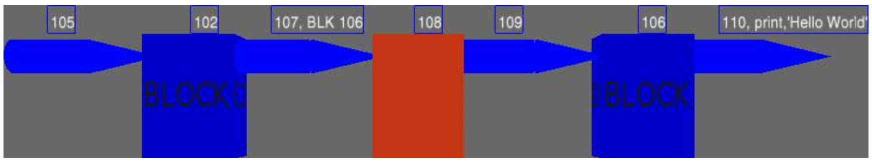
Instead, of explaining the visualisation in a step by step fashion similar to the 'Hello World' program §5.2.1, we will



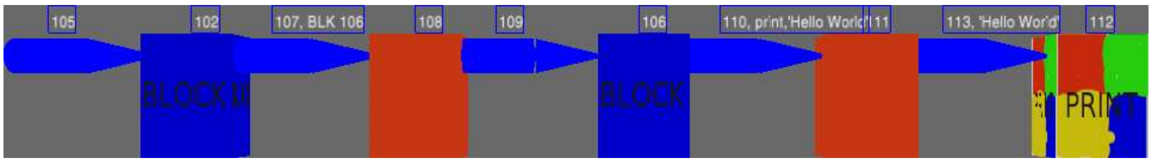
(a)



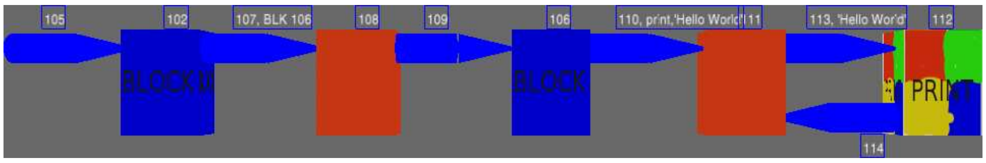
(b)



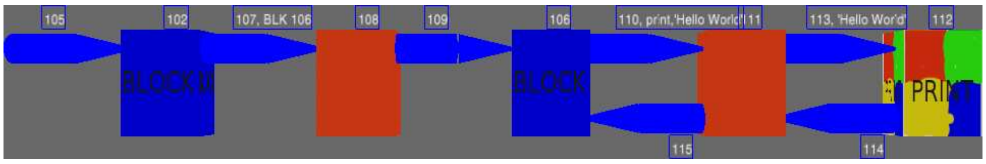
(c)



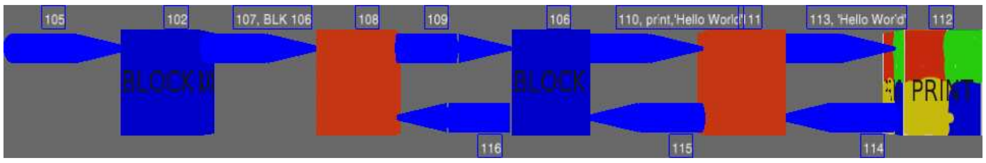
(d)



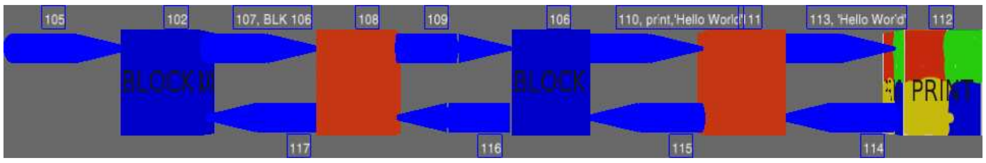
(e)



(f)



(g)



(h)

Figure 9: The above images are screenshots of the execution of the Hello World Program.

give an overview of it, highlighting any new aspects of visualisation.

- The first BlockConnect has 2 nested Connects which it executes in a serial fashion.The visualisation of the 2 lines of code is shown as 2 branches of the BlockConnect as shown in Figure 11(a).
- The visualisation of the BlockConnect *(= a 5)*, in the first line of code, is shown in Figure 11(b).
- The visualisation of the exact assignment operation (part of Figure 11(b)) is illustrated in Figure 11(c).
- Various stages of the assignment in Figure 11(c) are shown in Figure 12. You should have noticed by now that the Connect which wants to communicate with the symbol table sends out a signal with it's first element as 'arg'. Looking at 'arg' in the incoming signal, the GenericConnect sends the data to the symbol table in that local scope (see Figure 12(b)).
- The second line of code *(print a)* would be executed similar to the 'Hello World' program, except that the PrintConnect now resolves the value of the variable from the symbol table, before printing it on the terminal as shown in Figure 11(d).
- The PrintConnect gets the value of variable 'a', which is then printed onto the terminal. This is shown in Figure 11(e).

In short, this is how the variable assignment program executes. You should have already noticed that we have explained the visualisation in a top-down fashion, rather than the bottom-up fashion of the 'Hello World' program. Moreover, this top-down approach has enabled us to view understand the visualisation at various levels (line , block , expression, internal assignment). This flexibility in design, allows visualisation of code at various levels of granularity, which is further discussed in §5.4.5.

5.2.3 More examples

The visualisations of the above examples and a few more programs can be found online at:
http://users.boinc.ch/mansu/npl/npl.html.

The interested reader is encouraged to view the demo, to gain better insights into the visualisation than is possible with prose and some screenshots.

5.3 Implementation

The present system is a proof of concept prototype implemented in python. The libraries used for various modules are shown in Figure 2. Optimising for speed and space is part of future research. At present, the NPL programs are 16 to 45 times slower than an equivalent Python program[12]. However, note that the results can be off the mark, since the prototype is not optimised. The present implementation is cross platform[13].

[12]The performance figures are arrived at by taking average execution times of factorial of 1000 and the computing the first 1000 Fibonacci numbers.

[13]Runs on windows and Linux

5.4 Other Features of UI

The primary goal of NPL is to visualise the execution of code with its semantics intact. It was later realised that this design of NPL, based on COP, aids other applications which ease software development. Some of the potential applications are described below.

5.4.1 Customising the UI

Though the default visualisation of code is good enough to understand the execution of code, the programmer is usually lost in a sea of identical looking objects while visualising big blocks of code. To alleviate this problem, NPL allows its users to customise the appearance of the connects in the UI, paving way for domain specific visualisation. The visualisation can be customised by specifying a key : value pair in the ui.cfg file, containing the name of the function and a custom image to represent it in the UI, as shown below.

```
[textures]
fact:media/fact.png
```

The above configuration tells the visualiser to use the image fact.png whenever it displays the function fact. The visualisation without and with customisation is shown in Figure 13(a) and Figure 13(b) respectively. We can identify where the function is being called in Figure 13(b) in a quick glance, because we put out our visual cognition system to use.

One more example is the customised representation of the DictConnect used for storing the data of the symbol table as illustrated in Figure 5. Similarly, the look of any connect can be customised.

5.4.2 Interactive Testing and Debugging

Adding interaction capabilities to the Signals and Connects displayed in the NPL UI would double the UI as an interactive testing and debugging environment. By directly manipulating the objects in the UI, we can manipulate the data and control flow of the program in an intuitive fashion. For example, changing the data on the signal in Figure 14 from (+, 100, 98) to (−, 200, 98), we are altering the data(100 to 200) and control flow (addition to subtraction[14] operation) of the program.

The UI visualises and maintains a history of the execution of the program. This is useful while debugging, to answer the questions like "How did it get here?".

5.4.3 Code Visualisation at varying speeds

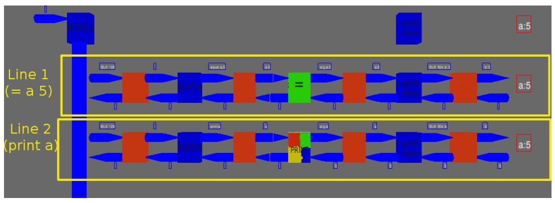
The speed with which a user can understand a visualisation depends on the person and on the complexity of the program. The NPL UI allows the user to vary the speed of visualisation while the program is running to accommodate users with varied skill levels.

5.4.4 Optimisations

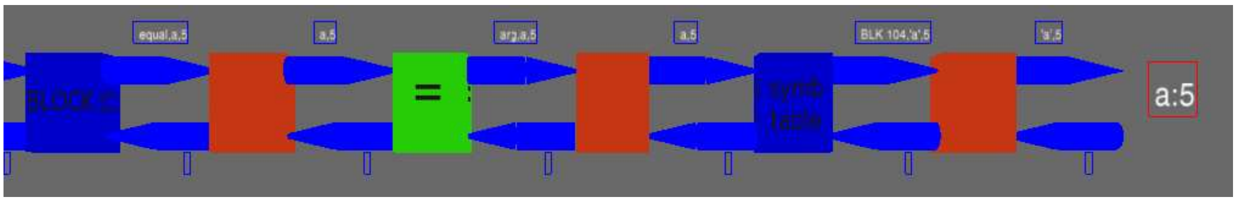
The motto of NPL is to show everything as it happens inside the interpreter, and this includes visualising the optimisations performed by the interpreter.

One example, of such an optimisation is GenericConnect creating the built-in connects without doing a lookup in the symbol table. This optimisation also helps in making the UI more intuitive.

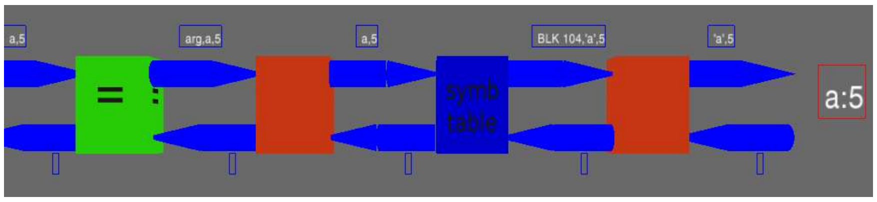
[14]can also be any valid function name



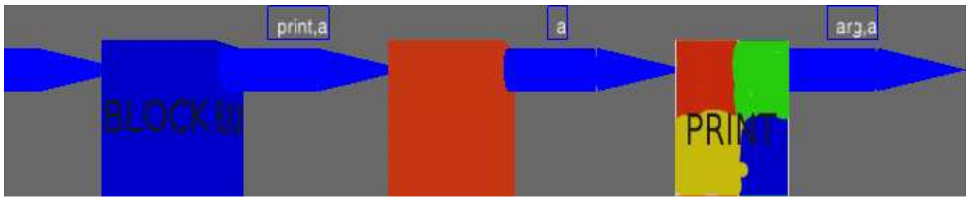
(a) The above image shows the execution of the 2 lines of the code in the UI



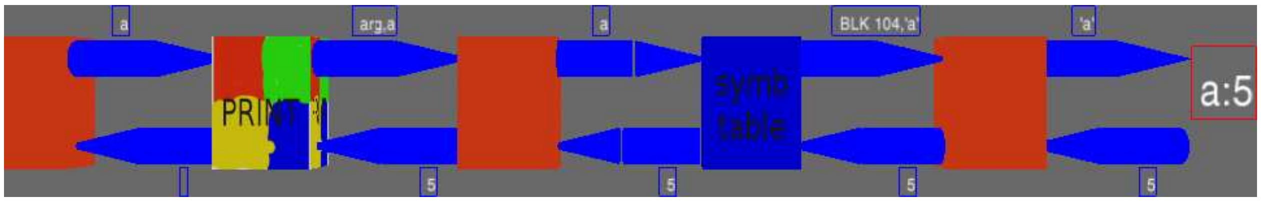
(b)



(c)

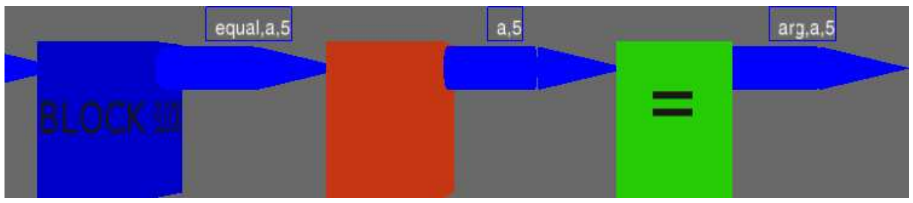


(d)

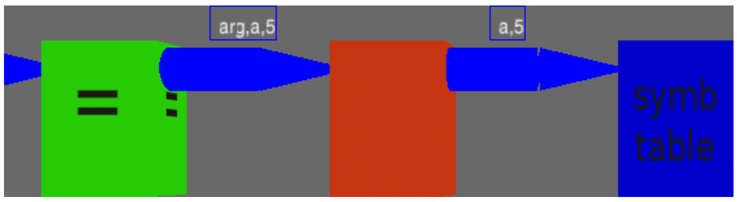


(e)

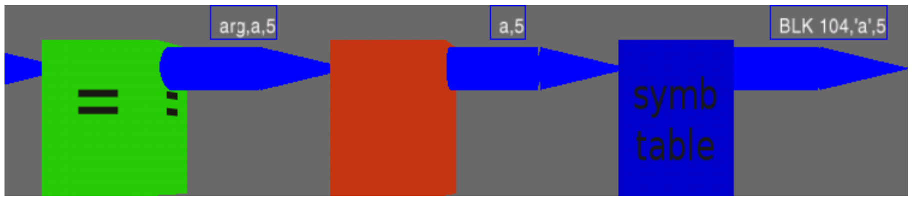
Figure 11: Some screenshots of the execution of the 'Variable Assignment' Program.



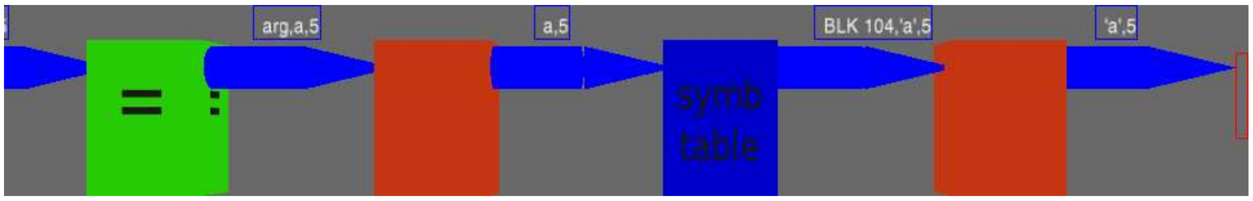
(a) EqualConnect sends the variable and it's value to the symbol table



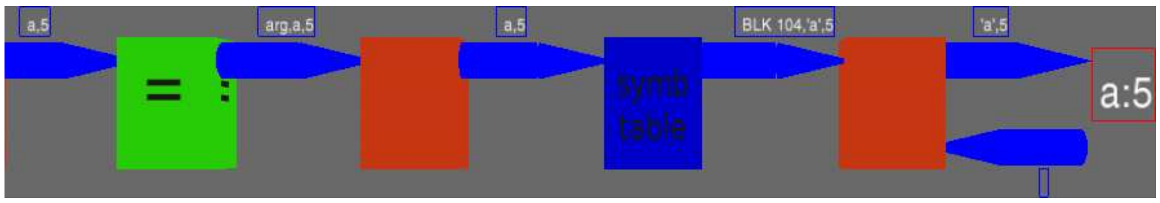
(b)



(c) Symbol Table delegates the actual task of storing the variable to a DictConnect

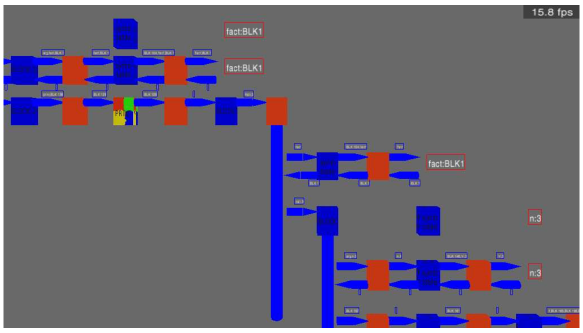


(d)

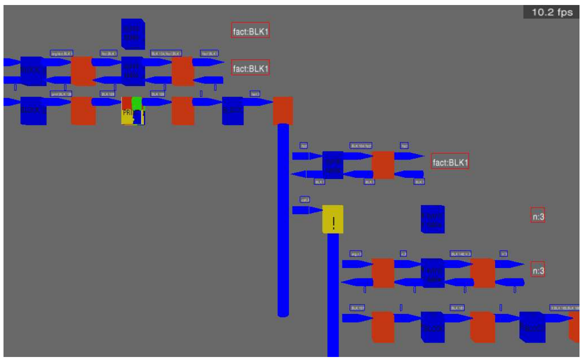


(e) The DictConnect saves the variable as a key:value pair

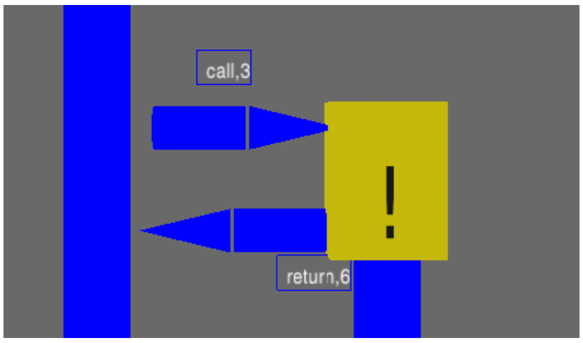
Figure 12: Screenshots of the execution of the assignment in the 'Variable Assignment' Program.



(a) UI without customisation



(b) UI with customisation



(c) Customised Connect showing Factorial function. The signals show the arguments and return values of the function.

Figure 13: The above images show the uncustomised(a) and customised(b) NPL UI for the execution of Factorial program. The Yellow box with '!' is the logic of the factorial function.

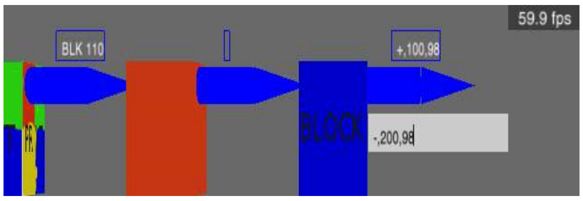


Figure 14: The Screenshot shows interactive editing of data on a Signal.

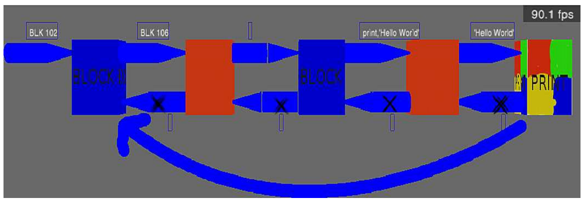


Figure 15: Screenshot of short-circuiting of Signals in "Hello World" program in Figure 8

You should have observed in the "Hello World" example that many of the connects were just returning the signal to their sender without doing any further processing on the data of the signal. This happens in approximately 50% of the signals. So, we have implemented an optimisation, in which the signal will be directly returned to the Connect which intends to do something with the data on the signal, instead of the immediate sender. So, in 'Hello World' example, the PrintConnect will return the signal directly to BlockConnect, instead of it's sender. This short-circuiting once implemented in UI, would be visualised as shown in Figure 15.

5.4.5 *Multiple Views of Code*

Providing multiple views of the source with varying levels of granularity[15] is the most wanted feature of any code visualisation system. Though the NPL UI does not provide these features in its current version, the present design of NPL language facilitates these features. Multiple views of code are possible in NPL by using special layout algorithms to layout Signals and Connects with custom visualisations.

The present design also allows viewing source at various levels of granularity. For example, a function level overview of the program would mean not drawing all the connects and signals on the UI after the BlockConnect which represents the function logic is drawn. In the factorial program above we will not draw anything beyond the yellow box with '!' symbol in Figure 13(b), if we need a function level overview. The input and output signals to this block represent the arguments passed to the function and the return value of the function respectively, as shown in Figure 13(c).

As we have seen, re-designing the language not only makes it easy to visualise code, but also makes way for other features which are difficult to implement in existing systems. Hence we feel that, *language is the foundation on which the tower of visualisation should be built.*

6. ADVANTAGES

[15]Immediate future goal of NPL UI

NPL is a better system for code maintenance as it is possible to visualise the execution of code with its semantics intact. The task can made easier further, by customising the visualisations and by viewing code at multiple levels of granularity.

Better visualisation also improves code quality (indirectly) because it enables more people to read code more easily and hence bugs can be caught faster. During development of the NPL system, we have caught a few bugs (harmless) in its implementation through the visualisation of test programs in the UI, which reinforces that visualisation can help us track bugs, which would not be caught by peer review of code or by test cases.

The simplicity and generality of COP makes it easy to visualise a subset of existing languages once their interpreters are re-designed based on COP. NPL is a proof of this as it's semantics are same as that of python language without the OO features.

The events that NPL interpreter produces are saved into a file, which can be used for off line visualisation of the execution of the program. The events generated by the interpreter do not have any UI information attached to them, hence better visualisers can be built without any modifications to the interpreter, thus making the visualisation implementation independent, which is one of our design goals.

In NPL the performance overhead due of visualisation is constant at 10%, since that is the time required to post everything there is about the execution of the program as events to the UI. For instrumented interpreters, the overhead usually increases with increasing granularity of the visualisation system.

7. DISADVANTAGES

At the first sight, NPL might not seem pragmatic because only code in NPL can be visualised. But we feel that long term advantages offered by a language similar to NPL far outweigh the short term cost of switching to NPL. Though it is possible to visualise existing languages by building interpreters for them using COP, they would not offer all the advantages compared to a language designed specifically for visualisation would offer.

This system cannot be used for visualising non-functional aspects of code like performance metrics etc. without further modifications to the interpreter. But the modifications needed would be minor.

As pointed out in [10], the utility of a visualisation system depends on the quality of visualisation. Better visualisers are needed for maintaining large bases of code and building better visualisers is not an easy task.

Though any feature can be added to NPL, we feel that some language features are easy to visualise compared to others. For example, anything performed at compile time (like c style macro expansion) will be difficult to visualise in this system.

8. RELATED RESEARCH

To the best of my knowledge, NPL is the first language designed with code visualisation as its design goal. However, many elements of our approach have a long history.

"Connect Oriented Paradigm", though independently designed, is very similar to the Actor Model [9] in which Actors send messages to each other. The Connect and Signal are analogous to Actor and Message respectively. Hence, NPL and ACT 1 [11][12] are similar in many respects. For example, ACT 1 has a distributed interpreter, consisting of a set of predefined actors which respond to messages which correspond to conventional actions of a interpreter. The NPL interpreter also provides the conventional facilities of an interpreter via various built-in connects. However, NPL is more specialised than ACT 1 as it is designed for visualisation. For example, the ACT 1 interpreter serialises the execution of events instead of a special actor analogous to BlockConnect in NPL. It cannot be hidden in NPL as we have to visualise the serial execution of the program.

The field of Visual Programming Languages sought to replace textual programming with diagrams [13]. But Visual Programming Languages did not work well in practice as diagrams cannot be as expressive as text [10][7]. Pictorial Janus [2] was different from its ilk as it unified the representation and execution of programs. However, you cannot see the code and its execution at the same time. But text is more expressive, precise and is sometimes faster to understand than a visual syntax.

Program Visualisation systems were built to visualise code. Most of the research in program visualisation has been either for teaching(algorithm animation) or performance tuning for applications, written in Java, C, C++, modula etc., but none of them make it easy for a developer to read code [10]. Algorithm animation systems like Balsa and Tango [10] though highly informative, are not suited for program visualisation and reading code as they require days to months to construct an animation. Developers in industry work on tight deadlines and do not have the luxury of building specialised animations. They need something like NPL which works out of the box and visualises code exactly as it was written, so that they can concentrate on the task at hand.

The Transparent Prolog Machine [4] provides an innovative graphical view of the program execution via an interface designed to suit Prolog's complex execution model. It was a sophisticated tracer with facilities for providing multiple views of source code with varying levels of granularity. However, it does not display the exact data and control flow of the program, but only few high level aspects of it [15]. Plater [14], a new SV system for Prolog was designed based on earlier research whose goal, in short, was to provide an overview of the execution program with the its semantics, but not the exact execution of code.

ZStep95 [8] was an interactive code stepper that supported reversible execution and animation of lisp programs. But it was a more sophisticated debugger than a code visualiser as a result it could only model a subset of LISP and the kinds of visualisation it produced were limited. NPL UI draws most of the inspiration, for the interactive testing and debugging of application via direct manipulation and correspondence of code with its execution, from this pioneering effort. NPL improves over it by visualising everything that happens in the interpreter in the UI instead of specific aspects of execution.

Though IDE's with integrated debugging facilities provide the facilities like linking code with the expression being evaluated, they are difficult to use and they do not understand the semantics of the language. Even a sophisticated debugger cannot visualise the optimisations, that are done in the interpreter.

Text is highly expressive. But animations are good at conveying abstractions and dynamism. NPL has the best of both worlds. We take the input program as text which is easier to express, has better density and more expressivity and show its execution as an animation while keeping the semantics of the program intact.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we have seen why a language should be redesigned to aid code visualisation. Next we have introduced the Connect Oriented Paradigm which was similar to the Actor Model upon which the NPL language was built. Based on COP we have designed the NPL Language and explained how the semantics of the language were implemented using the Signal and the 3 connects. Later we have seen a step by step visualisation of the "Hello World" program in the NPL UI. This example was a proof of the intuitive visualisation of code with its semantics intact.

We have also seen how this design of NPL aids in building domain specific visualisations of code and interactive testing and debugging of applications. These features are just tip of the iceberg and there is bright future ahead for NPL. Some of future goals of NPL are listed below:

- The ability to provide multiple views of the source code at various levels of granularity is the most important feature of any visualisation system.

- NPL at present only allows customising the look of an individual Connect. However, the ability to customise the look and the layout of a group of connects via advanced layout mechanisms will be a very useful feature for library builders who would like to ship a custom visualisation and layout for the components of their library along with the code. This feature will make languages with visualisation mainstream as they offer a clear advantage over existing approaches.

- The ability to interact with the program while supporting reversible execution via the UI will make testing and debugging systems a lot easier.

- Since the NPL interpreter is stack-less and is very much similar to the Actor model, we would like to add concurrency and distributed programming capabilities to NPL and the UI in future. OO facilities to NPL are also planned.

While working on NPL, I learned that the a careful balance of visualisation and language features would provide the best experience for programmers. We also observed that atleast a subset of existing languages can be visualised by re-designing the languages using COP. I would like to conclude that COP and visualisation may not effect the way we write and think about programs[16], but they would certainly effect the economics of writing programs.

[16]They may not change the semantics of existing languages in a significant way

10. REFERENCES

[1] E. S. M. B. A. Andres Moreno, Niko Myller. Visualising programs with jeliot 3. In *Proceedings of Advanced Visual Interfaces*. ACM, 2004.

[2] G. W. J. Q. Ch. Geiger, R. Hunstock. Visual modeling and 3d-representation with a complete visual programming language - a case study in manufacturing. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*. IEEE, September 1996.

[3] J. Edwards. Subtext: uncovering the simplicity of programming. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 505–518, New York, NY, USA, 2005. ACM Press.

[4] M. B. Eisenstadt M. Transperant prolog machine: An execution model and graphical debugger for logic programming. In *Journal of Logic Programming*, volume 5, pages 277–342, 1988.

[5] K. A. Frenkel. An interview with ivan sutherland. *Commun. ACM*, 32(6):712–714, 1989.

[6] A. M. Garcia. The design and implementation of intermediete codes for software visualisation. In *Master's Thesis,Department of Computer Science*. University of Joensuu, 2005.

[7] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[8] C. F. Henry Lieberman. Zstep95, a reversible, animated source code stepper. In *Software Visualisation: Programming as a Multimedia Experience*. MIT Press, September 1997.

[9] C. Hewitt. Viewing control structures as patterns of passing messages. In *AI Memo 410*. Artificial Intelligence Laboratoty,MIT, December 1976.

[10] M. H. B. Jhon Stasko, Jhon Domingue and B. A. Price. *Software Visualisation: Programming as a Multimedia Experience*. The MIT Press, Reading, Massachusetts, 1998.

[11] H. Lieberman. A preview of act 1. In *AI Memo 625*. Artificial Intelligence Laboratoty,MIT, June 1981.

[12] H. Lieberman. Concurrent object oriented programming in act 1. In *Object Oriented Concurrent Programming*. MIT Press, September 1987.

[13] T. L. Margaret Brunett, Adele Goldberg. *Visual Object-Oriented Programming:Concepts and Environments*. Prentice Hall, 1995.

[14] P. Mulholland. Using a fine-grained comparative evaluation technique to understand and design software visualization tools. pages 91–108.

[15] P. Mulholland. The effect of graphical and textual visualization on the comprehension of prolog execution by novices: an empirical analysis, 1994.

[16] N. Myller. The fundamental design issues of jeliot 3. In *Master's Thesis,Department of Computer Science*. University of Joensuu, 2004.

[17] W. P. on compilatioan Martin Rinard. http://www.ai.mit.edu/projects/dynlangs/talks/compilation-panel.htm. 2001.

[18] B. Verners. *Inside the Java Virtual Machine,2nd Edition*. McGraw Hill, http://www.artima.com/insidejvm/ed2/.