# Kathaa : NLP Systems as Edge-Labeled Directed Acyclic MultiGraphs

**Sharada Prasanna Mohanty**[1,2,3], **Nehal J Wani**[1],
**Manish Srivastava**[1], **Dipti Misra Sharma**[1]

[1] LTRC, International Institute of Information Technology, Hyderabad
[2] School of Computer and Communication Sciences , EPFL, Switzerland
[3] School of Life Sciences, EPFL, Switzerland
sharada.mohanty@epfl.ch, nehal.wani@research.iiit.ac.in
{m.shrivastava, dipti}@iiit.ac.in

## Abstract

We present **Kathaa**, an Open Source web-based Visual Programming Framework for Natural Language Processing (NLP) Systems. Kathaa supports the design, execution and analysis of complex NLP systems by visually connecting NLP components from an easily extensible Module Library. It models NLP systems an edge-labeled Directed Acyclic MultiGraph, and lets the user use publicly co-created modules in their own NLP applications irrespective of their technical proficiency in Natural Language Processing. Kathaa exposes an intuitive web based Interface for the users to interact with and modify complex NLP Systems; and a precise Module definition API to allow easy integration of new state of the art NLP components. Kathaa enables researchers to *publish their services* in a standardized format to enable the masses to use their services out of the box. The vision of this work is to pave the way for a system like Kathaa, to be the *Lego blocks* of NLP Research and Applications. As a practical use case we use Kathaa to visually implement the Sampark Hindi-Panjabi Machine Translation Pipeline and the Sampark Hindi-Urdu Machine Translation Pipeline, to demonstrate the fact that Kathaa can handle really complex NLP systems while still being intuitive for the end user.

## 1 Introduction

Natural Language Processing systems are inherently very complex, and their design is heavily tied up with their implementation. There is a huge diversity in the way the individual components of the complex system consume, process and spit out information. Many of the components also have associated services which mostly are really hard to replicate and/or setup. Hence, most researchers end up writing their own in-house methods for gluing the components together, and in many cases, own in-house re-implementations of the individual components, often inefficient re-implementations.

On top of that, most of the popular NLP components make many assumptions about the technical proficiency of the user who will be using those components. All of these factors clubbed together shut many potential users out of the whole ecosystem of NLP systems, and hence many potentially creative applications of these components. With Kathaa, we aim to separate the design and implementation layers of Natural Language Processing systems, and efficiently pack every component into consistent and reusable black-boxes which can be made to interface with each other through an intuitive visual interface, irrespective of the software environment in which the components reside, and irrespective of the technical proficiency of the user using the system. Kathaa builds on top of numerous ideas explored in the academia around Visual Programming Languages in general (Green and Petre, 1996) (Shu, 1988) (Myers, 1990), and also on Visual Programming Languages in the context of NLP (Cunningham et al., 1997). In a previous demonstration (Mohanty et al., 2016) at NAACL-HLT-2016, we showcased many of the features of Kathaa, and because of the general interest in Kathaa, we are now making an attempt to more formally model Kathaa in this paper.

This work is licenced under a Creative Commons Attribution 4.0 International Licence. Licence details: http://creativecommons.org/licenses/by/4.0/



*Proceedings of WLSI/OIAF4HLT*,
pages 1–10, Osaka, Japan, December 12 2016.

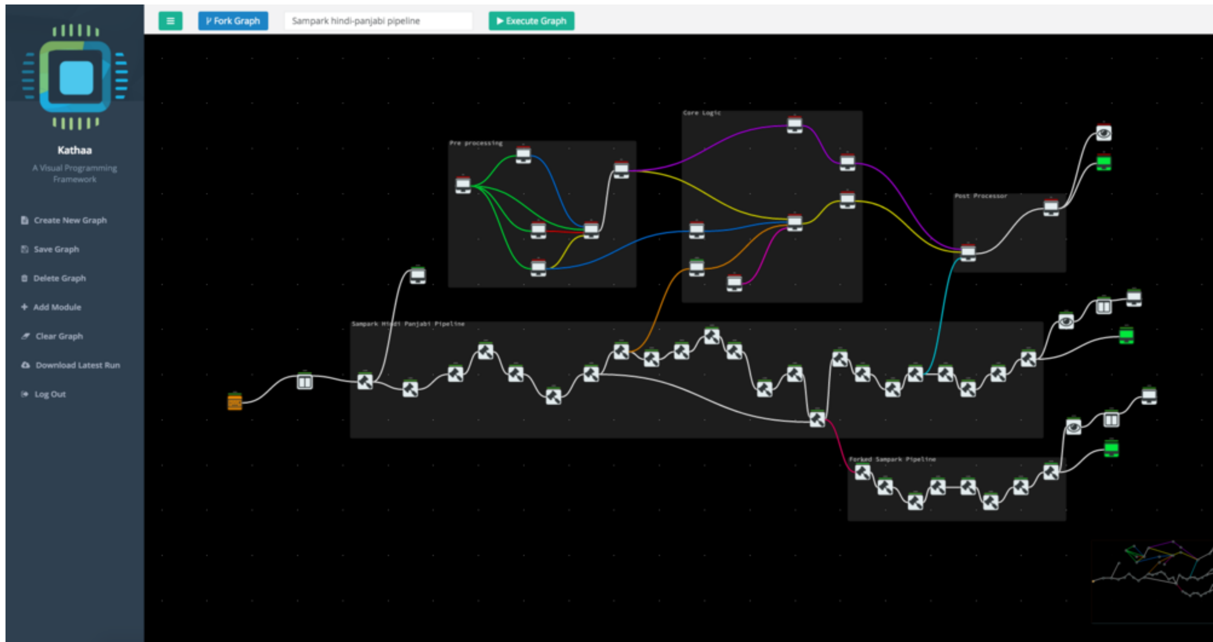


Figure 1: Example of a Hindi-Panjabi Machine Translation System, visually implemented using Kathaa.

## 2 Kathaa Modules

Kathaa Modules are the basic units of computation in the proposed Visual Programming Framework. They consume the input(s) across multiple input ports, process them, and finally pass on their output(s) across the many output ports they might have. The user has access to a whole array of such modules with different utilities via the Kathaa Module Library. The user can connect together these modules in any combination as he pleases (as long as the connections between the said modules are compatible with each other). The user also has the ability to tinker with the functionality of a particular module in real time by using an embedded code editor in the Kathaa Visual Interface during or before the execution of the Kathaa Graph. The inspiration behind Kathaa Modules comes from the Black Box approach in Integrated Digital Circuits, where rather complex combinations of Logic Gates are arranged in complex arrangements to get a desired behavior from the circuit, and then finally the overall arrangement is treated as a black box. The 'users' of this particular Integrated Circuit or 'chip', just need to refer to the Data Sheet of the chip for the input and output specifications to be able to use the said chip for designing more complex and high-level circuits.

Kathaa Modules aim to be the Integrated Circuits for NLP, which can be mixed and matched together to create interesting NLP Applications while hiding the complex implementation details in a black box.

### 2.1 Kathaa Data Blobs and Kathaa Data Sub Blobs

The input received at a particular input port, or the output generated at a particular output port of a module is always a collection of **kathaa-data-sub-blob**s. Each input port receives exactly the same number (as the rest of the input ports) of *kathaa-data-sub-blob*s, and similarly each output port holds exactly the same number (as the rest of the output ports) of *kathaa-data-sub-blob*s. A **kathaa-data-blob** is also a collection of *kathaa-data-sub-blob*s, but every *kathaa-data-blob* contains just one *kathaa-data-sub-blob* from each of the input ports (or the output ports, depending on if its a kathaa-data-blob at the Input Layer of the module or at the Output Layer of the module). The $n$-th *kathaa-data-sub-blob* of all the input ports, contributes to the $n$-th Input *kathaa-data-blob* of the Module Instance; similarly, the $m$-th *kathaa-data-sub-blob* of all the output ports, contributes to the $m$-th Output *kathaa-data-blob* of the Module Instance. A Kathaa Module has the capability to process multiple *kathaa-data-blobs* in parallel by spawning multiple instances of the same module during execution. The concept of numerous data blobs spread across multiple input ports (or output ports) enables us to efficiently empower module writers to leverage from

the inherent parallelizability in tasks performed by numerous NLP components. For example, some modules are parallelizable at the level of sentences, so if we have multiple sentences as inputs to this module, all those sentences are passed as different *kathaa-data-blob*s so that the framework can parallelize their processing depending on the availability of resources. Similarly, other modules could expect parallelizability at the level of words, or phrases or even a whole discourse. The *kathaa-data-blobs* were very much inspired by the data-blobs used in Caffe (Jia et al., 2014).

### 2.2 Formal definition of Kathaa Modules

A Kathaa Module can very simply be modeled as :

$$F(IP) = OP \tag{1}$$

where, $F()$ refers to the overall function the Module represents; $IP$ refers to the overall Input object that the Module receives; and $OP$ refers to the overall Output object that the Module produces.

The inputs and outputs for Kathaa Modules are spread across multiple input ports and output ports, so $IP$ and $OP$ can basically be represented as a collection of input and output values across all the input and output ports

$$IP = [IP_0, IP_1, ........., IP_{N-1}, IP_N] \qquad OP = [OP_0, OP_1, ........., OP_{M-1}, OP_M] \tag{2}$$

where $IP_n$ refers to the input received on input port $n$, where $n \in [0, N)$; and $N$ is the maximum number of Input Ports supported by Kathaa for Kathaa Modules; $OP_m$ refers to the output generated on output port $m$, where $m \in [0, M)$; and $M$ is the maximum number of Output Ports supported by Kathaa for Kathaa Modules.

Every input received on any of the input ports $IP_n$ or every output generated on any of the output ports $OP_m$ is but a collection of *kathaa-data-sub-blob*s, which are the basic primitives of data handling in Kathaa. So, $IP_n$ and $OP_m$ can be further represented as :

$$IP_n = [IP_{n,0}, IP_{n,1}, ........., IP_{n,X-1}, IP_{n,X}] \qquad OP_m = [OP_{m,0}, OP_{m,1}, ........., OP_{m,Y-1}, OP_{m,Y}] \tag{3}$$

Where $X$ and $Y$ refer to the number of *kathaa-data-sub-blob*s in each of the ports in the Input and Output layers respectively.

Now by substituting Equation 3 in Equation 2, we can represent $IP$ and $OP$ as a combined input and output matrices as follows :

$$IP = \begin{bmatrix} IP_{0,0} & IP_{0,1} & .. & P_{0,X-1} & IP_{0,X} \\ IP_{1,0} & IP_{1,1} & .. & P_{1,X-1} & IP_{1,X} \\ . & . & .. & . & . \\ . & . & .. & . & . \\ IP_{N-1,0} & IP_{N-1,1} & .. & IP_{N-1,X-1} & IP_{N-1,X} \\ IP_{N,0} & IP_{N,1} & .. & IP_{N,X-1} & IP_{N,X} \end{bmatrix} OP = \begin{bmatrix} OP_{0,0} & OP_{0,1} & .. & OP_{0,Y-1} & OP_{0,Y} \\ OP_{1,0} & OP_{1,1} & .. & OP_{1,Y-1} & OP_{1,Y} \\ . & . & .. & . & . \\ . & . & .. & . & . \\ OP_{M-1,0} & OP_{M-1,1} & .. & OP_{M-1,Y-1} & OP_{M-1,Y} \\ OP_{M,0} & OP_{M,1} & .. & OP_{M,Y-1} & OP_{M,Y} \end{bmatrix} \tag{4}$$

where $IP_{n,x}$ refers to the $x$-th *kathaa-data-sub-blob* on the $n$-th Input Port of the kathaa module; and $OP_{m,y}$ refers to the $y$-th *kathaa-data-sub-blob* on the $m$-th Output Port of the kathaa module.

Now lets define $IP_{*,k}$ as the collection of the $k$-th corresponding *kathaa-data-sub-blob*s across all the Input ports; and $OP_{*,p}$ as the collection of the $p$-th corresponding *kathaa-data-sub-blob*s across all the Output ports. So,

$$IP_{*,k} = [IP_{0,k}, IP_{1,k}, ........., IP_{N-1,k}, IP_{N,k}] \qquad OP_{*,k} = [OP_{0,k}, OP_{1,k}, ........., OP_{M-1,k}, OP_{M,k}] \tag{5}$$

$IP_{*,k}$ and $OP_{*,k}$ represent a single *kathaa-data-blob* in the Input and the Output layers respectively. Figure 2 shows the organization of *kathaa-data-blobs*, *kathaa-data-sub-blobs* and Input and Output Port values in the Combined Input and Output Matrices as defined in Equation 4.

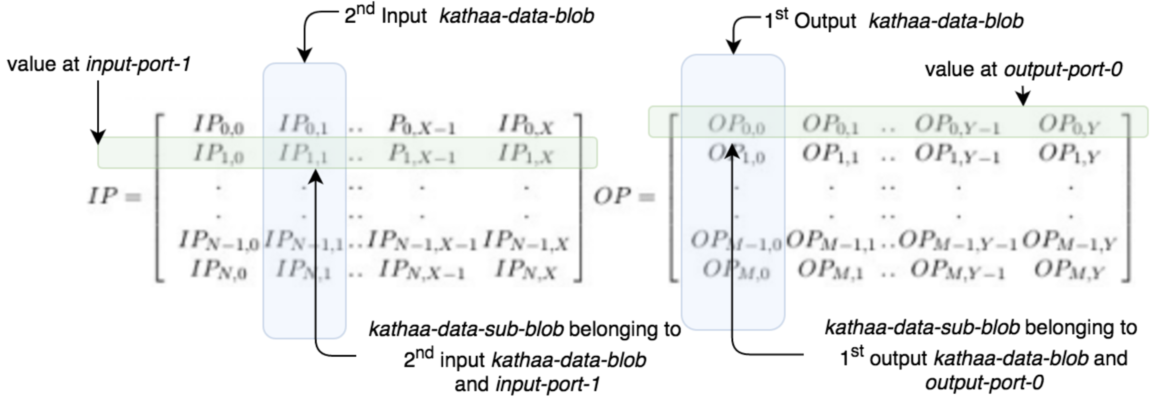


Figure 2: Organization of *kathaa-data-blobs*, *kathaa-data-sub-blobs* and Input and Output Port values in the Combined Input and Output Matrices as defined in Equation 4.

By using Equation 5, we can represent $IP$ and $OP$ as a collection of *kathaa-data-blob*s in the Input and Output layers as follows :

$$\begin{aligned} IP &= [(IP_{*,0})^T, (IP_{*,1})^T, .........., (IP_{*,X-1})^T, (IP_{*,X})^T] \\ OP &= [(OP_{*,0})^T, (OP_{*,1})^T, .........., (OP_{*,Y-1})^T, (OP_{*,Y})^T] \end{aligned} \tag{6}$$

Finally, substituting Equation 6 in Equation 1, we obtain :

$$\begin{aligned} F([(IP_{*,0})^T, (IP_{*,1})^T, .........., (IP_{*,X-1})^T, (IP_{*,X})^T]) = \\ [(OP_{*,0})^T, (OP_{*,1})^T, .........., (OP_{*,Y-1})^T, (OP_{*,Y})^T] \end{aligned} \tag{7}$$

For easy readability, we can define the substitutions $IB_i = (IP_{*,i})^T$ and $OB_i = (OP_{*,i})^T$, which represent an Input *kathaa-data-blob* and an Output *kathaa-data-blob* respectively. We can now rewrite Equation 7 as :

$$F([IB_0, IB_1, .........., IB_{X-1}, IB_X]) = [OB_0, OB_1, OB_2, .........., OB_{Y-1}, OB_Y] \tag{8}$$

which captures the crux of how Kathaa Modules manipulate data, by consuming a collection of *kathaa-data-blob*s, and finally spitting out a bunch of *kathaa-data-blob*s as Output.

### 2.3 Types of Kathaa Modules

#### 2.3.1 Kathaa General Modules

Kathaa General Modules are the class of Kathaa Modules which output the exact same number of *kathaa-data-blobs* as the number of *kathaa-data-blobs* in their Input Layer. Or more formally, in case of Kathaa General Modules, we will have $X = Y$ in Equation 8. As every Input *kathaa-data-blobs* independently maps to a single Output *kathaa-data-blobs*, we can imagine a function $f()$ which processes a single Input *kathaa-data-blobs*, to produce the corresponding Output *kathaa-data-blobs*.

$$\begin{aligned} F([IB_0, IB_1, .........., IB_{X-1}, IB_X]) &= [f(IB_0), f(IB_1), .........., f(IB_{X-1}), f(IB_X)] \\ &= [OB_0, OB_1, .........., OB_{X-1}, OB_X] \end{aligned} \tag{9}$$

Kathaa General Modules are the most common type of Kathaa Modules, and they are defined simply by defining the single function $f()$ which takes a single *kathaa-data-blob* as input, and finally returns a single *kathaa-data-blob* as output. The *kathaa-orchestrator* internally deals with the spawning of multiple instances of the said function $f()$ and processing all the Input *kathaa-data-blobs* in parallel, and then aggregating and writing their output as a collection of Output *kathaa-data-blobs*. The idea is that module developers just have to focus on the basic functionality of their module, and the efficient

parallelized execution of the same is automatically dealt by the framework, making it much easier for developers to get started with writing really powerful modules for Kathaa. To help developers get started, we also have a very flexible implementation of a `Custom Module` (Mohanty, 2016d) which can act as a quick starting point when defining Kathaa General Modules.

### 2.3.2 Kathaa Blob Adapters

**Kathaa Blob Adapters**, on the other hand, are a class of Kathaa Modules, which are provided with all the blobs from the Input Layer at the same time, and they have the ability to modify the number of blobs and pass it over to their output layer. They give the user a more fine grained control over the parallelizability of different parts of their Kathaa Graphs by varying the number of *kathaa-data-blob*s flowing through a particular point in the graph. More formally, Kathaa Blob Adapters are all the classes of Kathaa Modules where $X \neq Y$ in Equation 8. As a use case for Kathaa Blob Adapters, we can imagine a Kathaa Graph which receives a whole discourse as a single *kathaa-data-blob*, and it might benefit from processing the sentences parallely, and it could use a `Line Splitter` (Mohanty, 2016f) to split the whole discourse which was passed on as a single *kathaa-data-blob* into multiple *kathaa-data-blob*s each representing a single sentence, and when finally the user desired processing of the individual sentences are complete, a `Line Aggregator` (Mohanty, 2016e) could aggregate the processed sentences again into a single *kathaa-data-blob*. Similar kathaa-blob-adapters could be implemented to deal with splitting and aggregation of *kathaa-data-blob*s at the level of words, phrases,etc. For example, in contrast to the example cited above, if we are dealing certain language processing tasks which are inherently not parallalizable after a certain level of granularity, like, for e.g. Anaphora Resolution, Multi Document Summarisation, etc, the user will have to use an appropriate kathaa-blob-adapter, to make sure that all the information that is required for the particular task is available as a single *kathaa-data-blob*. In the case of Anaphora Resolution, a single *kathaa-data-blob* will contain a string of *N* sentences, and in the case of Multi Document Summarisation, a single *kathaa-data-blobs* will contain a string of *M* Documents.

### 2.3.3 Kathaa User Intervention Module

During the execution of a Kathaa Graph, the Kathaa Orchestrator ensures that the execution of a particular Kathaa Module Instance starts only after all its dependencies complete their execution and write their output. But in some NLP systems, the overall execution of the system might have to halt for some kind of user feedback. Like in the case of resource creation, where for example, you might want to start with a bunch of sentences, parse them using an available parser module, and then you would want to add Anaphora annotations by a human annotator (Sangal and Sharma, 2001). In that case, a **Kathaa User Intervention** Module can be used, where the overall execution at the particular Kathaa Module in the Kathaa Graph pauses till the user modifies the *kathaa-data-blobs* as she pleases and then resumes the execution at the said node. Internally, in case of Kathaa User Intervention Module, the Kathaa Orchestrator simply copies all the *kathaa-data-blobs* from the Input Layer to the Output Layer, and lets the user edit the Output Layer of the module through the Kathaa Visual Interface; and finally adds a User Feedback from the Kathaa Visual Interface as one of the dependencies of the Module. Kathaa Core Module group implements a `User Intervention` (Mohanty, 2016g) module, which can be used for the described use case in Kathaa Graphs.

### 2.3.4 Kathaa Resource Module

**Kathaa Resource Modules** are the class of Kathaa Modules which do not do any processing of the data, but instead they store and provide a corpus of text which can be used by any of the modules in the whole graph during execution. They do not have any Inputs, and they start executing right at the beginning of execution of the parent Kathaa Graph. More formally, they are the class of Kathaa Modules where $X = 0$ and $Y > 0$ in Equation 8.

### 2.3.5 Kathaa Evaluation Module

The aim of Kathaa is to provide an intuitive environment not only for prototyping and deployment but also debugging and analysis of NLP systems. Hence, we include a class of modules called as **Kathaa Evaluation Modules** which very much like kathaa-blob-adapters receive all the blobs across all the input

ports, do some analysis and spit out the results into the output ports. While in principle, this a subset of kathaa-blob-adapters, these modules enjoy a separate category among Kathaa Modules because of their utility in debugging and analyzing NLP systems. We implement a sample *Classification Evaluator* (Mohanty, 2016c) to help researchers quickly come up with easy to visualize confusion matrices to aid them in evaluating the performance of any of their subsystems. This could act as a starting point for easily implementing any other Evaluation Modules.

### 2.4 Kathaa Module Services

Most popular NLP Components work in completely different software environments, and hence standardizing the interaction between all of them is a highly challenging task. Kathaa can allow every module to define an optional service by referencing a publicly available *docker container* in the module definition. Kathaa can deal with the life-cycle management of the referenced containers on a configurable set of Host Machines. The corresponding kathaa-modules function definition then acts as a light weight wrapper around this service. This finally enables different research groups to **publish their service** in a consistent and reusable way, such that it fits nicely in the Kathaa Module ecosystem.

### 2.5 Kathaa Module Packaging and Distribution

By design, the definition of Kathaa Modules is completely decoupled from the actual code base of the Kathaa Framework. The idea was to facilitate the possibility of a large community of independent and unsupervised contributors and a swarm of community contributed modules which would ultimately be available by a simple to use Kathaa-Package-Manager. While the Kathaa-Package-Manager is not yet implemented, we believe it would be pretty trivial to implement the same because of the way Kathaa Modules are designed. A set of related Kathaa Modules reside as a Kathaa Module Group in a publicly accessible `git` Repository, and the Kathaa Framework downloads and loads these Modules on the fly just by referencing their publicly accessible `URI` in the overall configuration file. Detailed documentation on the definition, packaging and distribution of Kathaa Modules along with a list and description of all the available Kathaa Modules can be found online at : *Kathaa Module Packaging and Distribution* (Mohanty, 2016b).

## 3 Kathaa Graph

A Kathaa Graph is an edge-labeled Directed Acyclic MultiGraph of 'instances' of Kathaa Modules, with Edges connecting one or more Output Ports of one instance of a Kathaa Module to one or more Input Ports of an instance of the same or different Kathaa Module. A Kathaa Graph can have multiple instances of the same Kathaa Module at different positions in the graph, and also with different configurations of the said instances. Each Module Instance maintains its own state, and there can be multiple directed edges between any two Module Instances. We start by defining a few key variables:

- $M_k$ represents a Kathaa Module Instance, where $k \in [0, K)$ and $K$ is the total number of Kathaa Module Instances in a Kathaa Graph $G$.
- $\psi(M_k)$ represents the state of a Kathaa Module Instance, where state being all configurable parameters of the Module, a copy of the data present at all the Input and the Output ports and the definition of the associated computational function.
- $\tau(M_k)^I$ represents the set of all input ports of the Kathaa Module Instance $M_k$
- $|\tau(M_k)^I|$ represents the cardinality of the set $\tau(M_k)^I$ and hence represents the total number of Input Ports in the Module Instance $M_k$
- $\tau(M_k)^I_i$ represents the $i$-th input port of the Kathaa Module Instance $M_k$
- $\eta(\tau(M_k)^I_i)$ represents the total number of incoming edges into the $i$-th input port of the Kathaa Module Instance $M_k$
- $\tau(M_k)^O$ represents the set of all output ports of the Kathaa Module Instance $M_k$
- $|\tau(M_k)^O|$ represents the cardinality of the set $\tau(M_k)^O$ and hence represents the total number of Output Ports in the Module Instance $M_k$

- $\tau(M_k)^O_j$ represents the $j$-th output port of the Kathaa Module Instance $M_k$
- $\eta(\tau(M_k)^O_j)$ represents the total number of outgoing edges from the $j$-th output port of the Kathaa Module Instance $M_k$

(10)

We define a Kathaa Graph $G$ as an ordered pair of its vertices $V$ and edges $E$:

$$G = (V, E) \tag{11}$$

The Vertices of a Kathaa Graph are composed of Kathaa Module Instances, so we can write

$$V = \{M_k | k \in [0, K)\} \tag{12}$$

There can be multiple edges between two Nodes in a Kathaa Graph, for example when you add a directed edge between the output-port-1 of a Kathaa Module instance $M_1$ to the input-port-1 of another Kathaa Module instance $M_2$. Then you add another directed edge between the output-port-2 of the Kathaa Module Instance $M_1$ to the input-port-2 of the Kathaa Module instance $M_2$. Or more formally, Kathaa Graphs can have parallel edges.

So we define an Edge $e$ in a Kathaa Graph $G$ as an ordered 3-tuple :

$$e = (s, t, L) \tag{13}$$

Where, $s$ ($s \in V$) is the Source Node of the edge; $t$ ($t \in V$) is the Target Node of the edge, and $L$ is the edge-label of the edge $e$ which is represented as an ordered tuple

$$L = (s_{OP}, t_{IP}) \tag{14}$$

- $s_{OP}$ refers to the output port of the Source Node of the edge $e$. And $s_{OP} \in \tau(s)^O$
- $t_{IP}$ refers to the input port of the Target Node of the edge $e$. And $t_{IP} \in \tau(t)^I$

Substituting Equation 14 in Equation 13 we obtain :

$$e = (s, t, (s_{OP}, t_{IP})) \tag{15}$$

Now the set of edges $E$ of the Kathaa Graph $G$ can defined as :

$$E = \{(s, t, (s_{OP}, t_{IP})) \mid \ s, t \in V \wedge s \neq t \wedge S_{OP} \in \tau(s)^O \wedge t_{IP} \in \tau(t)^I \wedge \kappa(s_{OP}, t_{IP}) \wedge \eta(t_{IP}) = 1\} \tag{16}$$

Where, $s, t$ are defined in Equation 13; $s_{OP}, t_{IP}$ are defined in Equation 14; $\tau()^O$ and $\tau()^I$ and $\eta()$ are defined in Equation 10; and $\kappa(s_{OP}, t_{IP})$ refers to a boolean function which determines the compatibility of a particular Input port and Output port pair based on meta structure definition of the corresponding modules.

The conditions in Equation 16 represent some of the key properties of a Kathaa Graph. $s, t \in V$ asserts that both the Source Node and the Target Node have to be from the set of Nodes or the Module Instances in the Kathaa Graph. $s \neq t$ asserts that self loops are not allowed in a Kathaa Graph, so the Source Node and the Target Node in a Kathaa Graph cannot be the same. $S_{OP} \in \tau(s)^O$ asserts that the Output Port from the Source Node that is associated with an edge has to be a valid Output Port from the set of Output Ports of the Source Node. $t_{IP} \in \tau(t)^I$ asserts that the Input Port from the Target Node that is associated with an edge has to be a valid Input Port from the set of Output Ports of the Target Node. $\kappa(s_{OP}, t_{IP})$ asserts that the Output Port from the Source Node and the Input Port at the Target Node have to be compatible with each other based on the meta structure definition of the Source and Target Nodes. $\eta(t_{IP}) = 1$ asserts that there can be only a single edge that can be associated with an Input Port of any Module Instance in the Kathaa Graph. Now, Equation 16 defines $E$ and Equation 12 defines $V$, hence we can use them in Equation 11 to finally be able to formally define a Kathaa Graph $G$.

## 4 Kathaa Orchestrator

Kathaa Orchestrator obtains the structure of the Kathaa Graph and the initial state of the execution initiator modules from the Kathaa Visual Interface, and then it efficiently orchestrates the execution of the graph depending on the nature and state of the modules, while dealing with process parallelisms, module dependencies, etc under the hood. The execution of a Kathaa Graph starts by collecting all the Nodes in the Kathaa Graph from which it should start the execution. In the case that the Visual Interface provides a Module Instance to begin execution from, the Kathaa Graph starts execution from just that Node, else it collects all the `sentence_input` nodes and the `resource` nodes. The collected nodes are simply queued in a global Job Queue. A background process, in the meantime, listens on the Job Queue, and whenever a Job is added to the queue, it tries to execute the Job on any of the available resources. The execution of the Job starts by trying to execute the actual function associated with the particular Node, and if its successful, it passes the data along all its outgoing edges to the designated input ports of module instances further along the graph, and then finally returns the obtained result object. A Job Complete event handler is called with the final result (or the exception in case of errors), and the Job Complete event handler passes along the data to the Visual Interface to update the state of the graph in the Visual Interface and provide the user with the result associated with the Module or the actual exception and the error message to help the user debug the particular Kathaa Graph. Detailed documentation on Kathaa Orchestrator can be found at : *Kathaa Orchestrator* (Mohanty, 2016a).

## 5 Kathaa Visual Interface

Kathaa Interface lets the user design any complex NLP system as an edge-labeled Directed Acyclic MultiGraph with the Kathaa Module Instances as nodes, and edges representing the flow of *kathaa-data-blobs* between them. Users have the option to not only execute any such graph, but also interact with it in real time by changing both the state and functionality of any of the module right from within the interface. It can be a really useful aid in debugging complex systems, as it lets the User easily visualize and modify the flow of kathaa-data-blobs across the whole Kathaa Graph. Apart from that, it also encourages code-reuse by lettings users 'Fork' a graph, or 'Remix' the designs of NLP systems to come up with better and adapted versions of the same systems. Figure 1 shows the visual implementation of a Hindi Panjabi Machine Translation system in the Kathaa Visual Interface.

## 6 Use Cases

Kathaa, as a Visual Programming Framework was developed with Sampark Machine Translation System as a use case. We ported all the modules of the *Hindi-Panjabi* (Mohanty, 2016h) and *Hindi-Urdu* (Mohanty, 2016i) Translation Pipelines of Sampark Machine Translation System(SAM, 2016) into Kathaa. We then demonstrated the use of Kathaa in creation of NLP Resources by the use of Kathaa User Intervention modules, and also moved on to demonstrate visual analysis of different classification approaches by using the Kathaa-Classification-Evaluation module. We are currently also exploring the use of Kathaa in classrooms to help students interact with and design complex NLP systems with a much lower barrier to entry. All these example Kathaa Graphs are the seed Graphs that are included in the repository, and can be used out of the box. It is important to note that these use cases that we managed to explore are only the tip of the iceberg when it comes to what is possible using a framework like Kathaa. One of the key features in Kathaa which enables for it to be used in a whole range of use cases is the easy extensibility. The Kathaa Module Definition API, enables the user of the system to theoretically define any function as a Kathaa Module. Also, Kathaa internally works using event triggers, hence making it a practical possibility to define modules which may run for days or weeks, quite helpful when exploring Kathaa for use cases where the user might want to define a Kathaa Module which trains a model based on some pre-processed data. The NPM (Tilkov and Vinoski, 2010) inspired packaging system, is again something which we believe can help with large scale adoption of a system like Kathaa. It paves the way for a public contributed repository of NLP components, all of which can be mashed together in any desired combination. The ability to optionally package individual services using Docker Containers

also helps make a strong case when pitching for the possibility of a large public contributed repository of NLP components. These are a few things which set Kathaa apart from already existing systems like LAPPS Grid (Ide et al., 2014), ALVEO (Cassidy et al., 2014) where the easy extensibility of the system is a major bottleneck in its large scale adoption. The inter-operability between existing systems is also of key importance, and the design of Kathaa accommodates for its easy adaptation to be used along with other similar system. The assumption, of course, is that a wrapper Kathaa Module has to be designed for each target system using the Kathaa Module Definition API. The wrapper modules would be completely decoupled from the Kathaa Core codebase, and hence can be designed and implemented by anyone just like any other Kathaa Module.

A demonstration video of many features and use cases of Kathaa is also available to view at :
`https://youtu.be/woK5x0NmrUA`

## 7 Conclusion

We demonstrate an open source web based Visual Programming Framework for NLP Systems, and make it available for everyone to use under a MIT License. We hope our efforts can, in some way, catalyze more new and creative applications of NLP components, and enables an increased number of researchers to more comfortably tinker with and modify complex NLP Systems.

## Acknowledgements

The first real world implementation of a Kathaa Graph was achieved by porting numerous modules from Sampark MT system developed during the "Indian Language to Indian Language Machine translation" (ILMT) consortium project funded by the TDIL program of Department of Electronics and Information Technology (DeitY), Govt. of India. Kathaa is built with numerous open source tools and libraries, an (almost) exhaustive list of which is available in the Github Repository of the project, and we would like to thank each and every contributor to all those projects.

## References

Steve Cassidy, Dominique Estival, Tim Jones, Peter Sefton, Denis Burnham, Jared Burghold, et al. 2014. The alveo virtual laboratory: A web based repository api.

Hamish Cunningham, Kevin Humphreys, Robert Gaizauskas, and Yorick Wilks. 1997. Gate - a general architecture for text engineering. In *Proceedings of the Fifth Conference on Applied Natural Language Processing: Descriptions of System Demonstrations and Videos*, pages 29–30, Washington, DC, USA, March. Association for Computational Linguistics.

Thomas R. G. Green and Marian Petre. 1996. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *J. Vis. Lang. Comput.*, 7(2):131–174.

Nancy Ide, James Pustejovsky, Christopher Cieri, Eric Nyberg, Di Wang, Keith Suderman, Marc Verhagen, and Jonathan Wright. 2014. The language application grid. In *LREC*, pages 22–30.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA. ACM.

Sharada Prasanna Mohanty, Nehal J. Wani, Manish Shrivastava, and Dipti Misra Sharma. 2016. Kathaa: A visual programming framework for NLP applications. In *Proceedings of the Demonstrations Session, NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12-17, 2016*, pages 92–96.

Sharada Prasanna Mohanty. 2016a. Kathaa Documentation : Kathaa Orchestrator. `https://github.com/kathaa/kathaa/blob/master/docs/kathaa-orchestrator.pdf`. [Online; accessed 16-October-2016].

Sharada Prasanna Mohanty. 2016b. Kathaa Documentation : Module Packaging and Distribution. `https://github.com/kathaa/kathaa/blob/master/docs/ModulePackagingAndDistribution.pdf`. [Online; accessed 16-October-2016].

Sharada Prasanna Mohanty. 2016c. Kathaa Module : Classification Evaluator. https://git.io/vV40f. [Online; accessed 16-October-2016].

Sharada Prasanna Mohanty. 2016d. Kathaa Module : Custom Module. https://git.io/vV4Rp. [Online; accessed 16-October-2016].

Sharada Prasanna Mohanty. 2016e. Kathaa Module : Line Aggregator. https://git.io/vV40v. [Online; accessed 16-October-2016].

Sharada Prasanna Mohanty. 2016f. Kathaa Module : Line Splitter. https://git.io/vV4Rj. [Online; accessed 16-October-2016].

Sharada Prasanna Mohanty. 2016g. Kathaa Module : User Intervention. https://git.io/vV40U. [Online; accessed 16-October-2016].

Sharada Prasanna Mohanty. 2016h. Kathaa Module Group : Sampark Hindi Panjabi Translation Pipeline Modules. https://github.com/kathaa/hindi-panjabi-modules. [Online; accessed 16-October-2016].

Sharada Prasanna Mohanty. 2016i. Kathaa Module Group : Sampark Hindi Urdu Translation Pipeline Modules. https://github.com/kathaa/hindi-urdu-modules. [Online; accessed 16-October-2016].

Brad A. Myers. 1990. Taxonomies of visual programming and program visualization. *J. Vis. Lang. Comput.*, 1(1):97–123.

2016. Sampark: Machine translation among indian languages. http://sampark.iiit.ac.in/sampark/web/index.php/content. Accessed: 2016-02-10.

Rajeev Sangal and Dipti Misra Sharma. 2001. Creating language resources for nlp in indian languages 1. background.

Nan C Shu. 1988. *Visual programming*. Van Nostrand Reinhold.

Stefan Tilkov and Steve Vinoski. 2010. Node. js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80.